2008

Investigations of cellular automata-based stream ciphers

Joseph S. Testa

Investigations of Cellular Automata-based Stream Ciphers

by

Joseph S. Testa II

A Thesis

Submitted to the Faculty

of the

ROCHESTER INSTITUTE OF TECHNOLOGY

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Security and Information Assurance

by

_____

May 12, 2008

APPROVED:

_____
Professor Alan Kaminsky, Thesis Advisor

_____
Professor Hans-Peter Bischof, Reader

_____
Professor Peter Lutz, Observer

ABSTRACT

In this thesis paper, we survey the literature arising from Stephan Wolfram's original paper, “Cryptography with Cellular Automata” [WOL86] that first suggested stream ciphers could be constructed with cellular automata. All published research directly and indirectly quoting this paper are summarized up until the present. We also present a novel stream cipher design called Sum-4 that is shown to have good randomness properties and resistance to approximation using linear finite shift registers. Sum-4 is further studied to determine its effective strength with respect to key size given that an attack with a SAT solver is more efficient than a brute-force attack. Lastly, we give ideas for further research into improving the Sum-4 cipher.

TABLE OF CONTENTS

1. INTRODUCTION

Cryptography is a central component of information security. Financial and medical information, business, political, and military strategies, and personal secrets all require cryptography in order to maintain secrecy and integrity while in transit over untrusted communication mediums. Cryptography is critical in maintaining order in our current information-based society; without strong barriers protecting information and controlling access to it, society would surely descend into chaos.

In order for cryptography to be meaningful, it must be carried out in a unique way, much like the security of a lock depends upon the fact that only authorized parties have the corresponding key; if all locks used the same key, then a thief need only buy one lock to get the key for all locks. While it is clear that uniqueness is certainly a major concern for cryptographic systems, randomness is of equal importance. Should a cryptographic system use keys with sequential ordering, then if one single key is discovered by an attacker, then all the keys (both past and future) can be deduced. Thus, both uniqueness and randomness are essential components to any cryptographic system. We study both properties in this paper.

A cipher, or encryption function, either works upon a set number of bytes at a time (called a *block cipher*), or by encrypting individual bits (called a *stream cipher*). While

block ciphers have enjoyed many years of practical use due to the United States government's standardizations of the DES [NIS99] and AES [NIS01c] block ciphers, in this paper we will focus on stream ciphers. While DES and AES have never been broken, other systems over the years that were once thought to be secure have crumbled. For this reason, it is prudent to invest efforts in redundant cryptographic systems as an application of the wise defense-in-depth security principle. Therefore, research into lesser-used stream ciphers remains highly relevant.

In this paper, we survey the literature arising from Wolfram's original paper in 1986 [WOL86] that suggested that *cellular automata* can be used to generate a key-stream for a stream cipher. In chapters 2 through 8, we summarize all papers that describe any cryptographic primitive constructed using cellular automata. In chapter 9, we propose a novel design for a stream cipher called Sum-4 using cellular automata and analyze its security properties. In chapter 10, we give ideas for future work on the Sum-4 cipher. Lastly, the paper is concluded in chapter 11.

1. 1 Definition of Cellular Automata

Cellular automata (CA) are finite state machines. They are composed as a set of cells, a_i, linked in space that are updated synchronously in time based on an update

function, *f* (sometimes called a *transition function* or *rule*). One of the most basic forms of cellular automata are the so-called Wolfram cellular automata (WCA), which are characterized by $a_i^{t+1} = f(a^t_{i-1}, a_i^t, a^t_{i+1})$, where a_i^t is the cell at position *i* at time *t*. Each cell is updated by incorporating itself and its left and right neighbors into *f*, which determines its value in the next time step.

As each cell can hold two values (0 or 1), and that the update function takes three cells as input, *f* can operate upon $2^3 = 8$ different combinations of inputs. Thus, there are $2^{2^3} = 256$ different kinds of outputs for those inputs (i.e.: there are exactly 256 possible rules for WCA). Each rule can be expressed as a decimal number, which represents its output given all 8 possible inputs:

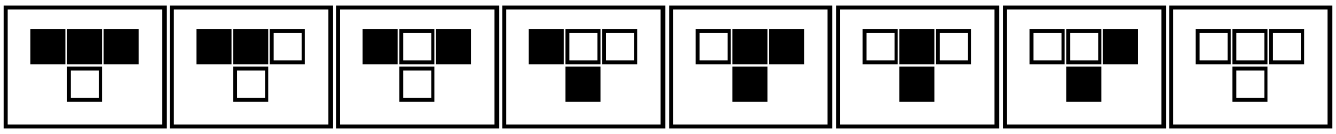

Figure 1.1. The outputs of rule 30 given its inputs. The squares each represent one cell. A white square is a cell with value 0; a black square is a cell with value 1. The first row of three cells is the rule's input, and the single cell below it is the output. Rule 30 is thus given its name because of its output bits: 00011110 in binary is 30 in decimal.

WCA are one-dimensional because the cells are linked to left and right neighbors only. Thus, they can be conceived as forming a straight line. Other types of CAs that are actively studied are two-dimensional CAs, where the cells also have top and bottom neighbors, and are possibly connected to cells on the diagonal; these can be conceived as forming a two-dimensional grid.

Because practical CAs are finite in space, they must have extremities. Those extremities lack the full number of neighbors that the update function requires (i.e.: the leftmost cell in a one-dimensional CA does not itself have a left neighbor). There are several approaches to handle this situation. A CA is said to have a *null boundary* when its extremities are hard-coded to have neighbors with value of 0. It said to have a *periodic*, or *cyclical boundary* if the extremities are linked to each other.

Below is an illustration of a WCA with 256 cells updated with rule 30 having a cyclical boundary and a random initial state:

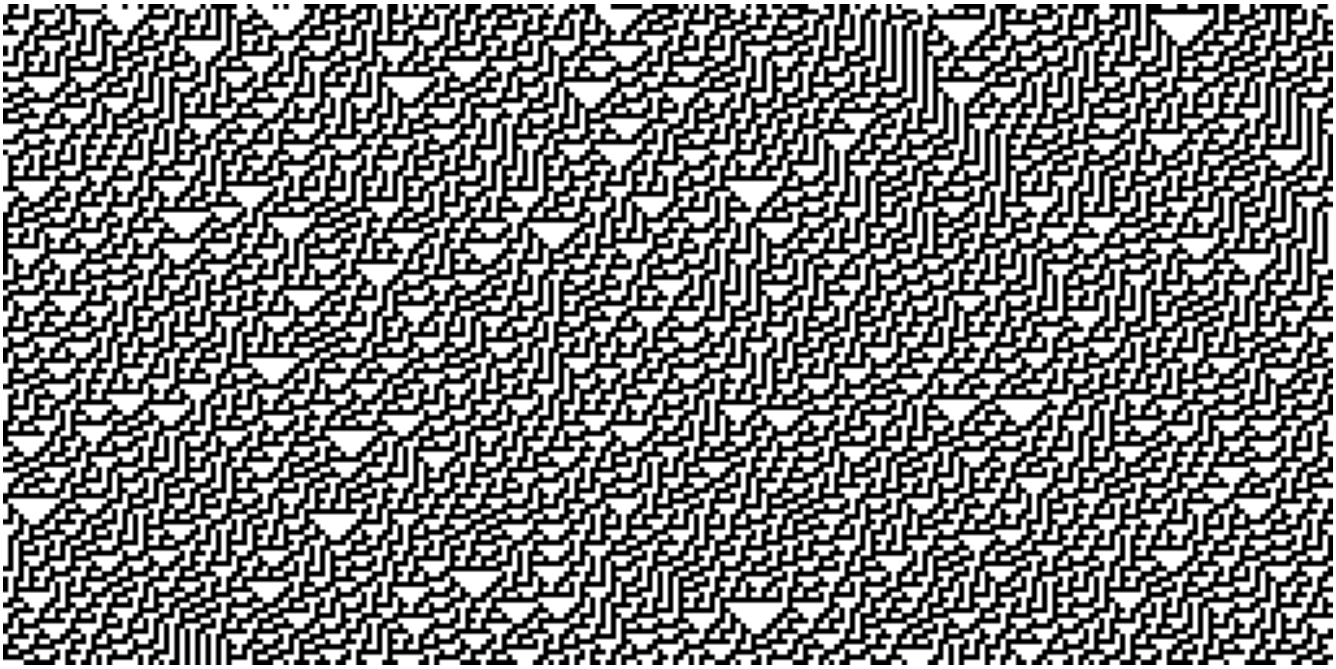

Figure 1.2. The top row represents the cells in their initial state (chosen randomly). Subsequent rows represent those same cells after rule 30 has been applied to the row above it.

CAs are not necessarily restricted to two immediate neighbors (i.e.: a radius of 1). Work has been done to analyze the properties of CAs with a radius of 2 and 3 (see [SER04], [BOU04], [MAR03b]). However, increasing the radius size too much destroys the benefits of *locality*, which results in a more expensive hardware implementation.

A major advantage of CAs is that the cells can be updated entirely in parallel to each other; they only depend upon the value of three previous cells. Thus, they can be

implemented very efficiently in hardware, with all cells updated synchronously with a single clock cycle. In software, their parallelism can result in an efficient multi-threaded implementation that can take advantage of today's multiple-core CPUs.

1.2 Wolfram's Original Paper

Stephen Wolfram first proposed in 1986 that one-dimensional cellular automata could be used as a cryptographic primitive [WOL86]. He suggested that the state of the CA can be initialized using a key, and that the output of a single cell through time is cryptographically random for a certain equation ($a_i' = a_{i-1}$ XOR (a_i OR a_{i+1})) (also known as rule 30), thus making the scheme suitable for use as a key-stream generator. Furthermore, he suggests that, given knowledge of an output stream, the problem of deducing the initial state is in the class NP (since the problem of boolean satisfiability is in class NP).

Ever since its publication, this paper has sparked an interest in the cryptographic community for using CAs as primitives in stream ciphers, block ciphers, hash functions, message authentication codes, and public key cryptosystems. The literature arising from Wolfram's paper to the present is detailed in chapters 2 through 8.

2. GENERAL LITERATURE

In this chapter, we examine general literature that has given interesting results relating to CAs and cryptography. Several papers discuss how to use genetic programming to locate optimal rules for generating random numbers. Others in this chapter describe how to obtain CAs with certain properties such as maximal diffusion of bits from an initial state, or how to obtain a CA with maximal period.

Sipper and Tomassini (1996) present a method for constructing a co-evolving algorithm for cellular programming, resulting in a non-uniform CA [SIP96]. Their co-evolving technique differs from previous genetic programming results from other researchers [TOM95] mainly in that the fitness of each cell is dependent upon its neighbors, instead of operating wholly upon independent populations and applying the genetic operators only after a CA is computed completely. They present pseudo-code of the algorithm used to compute and evaluate rule maps, along with two specific rule maps that were generated by it. Statistical analysis for randomness is presented on the co-evolved rule maps that show that they are at least as good as all previously-known CA randomness output (via Wolfram's rule 30 [WOL86] and mixtures of rules 90 and 150 presented in [HOR89a] and [HOR89b]).

Koc and Apohan (1997) present an inversion algorithm use for cryptanalysis of

CA key generators [KOC97]. Given an n-bit state of a rule 30 CA, the algorithm can calculate the initial state in $O(n)$ steps for certain seeds, or at most $O(2^{n/2})$ for arbitrary initial states. More importantly, their algorithm can calculate any initial state from any one-dimensional CA in at most $2^{(q-1)(1-a)n}$ steps, where q is the number of neighbors and a is the probability of agreement between the update function and the best affine function that approximates it. The authors note that their algorithm can be applied in conjunction with Meier and Staffelbach's attack [MEI91] in the following way: Meier and Staffelbach's algorithm is first used on a bit stream to calculate one complete row in the CA state, then Koc and Apohan's algorithm is used to invert the completed row so that the key is found. The authors' algorithm shows that, to suggest a new key-stream generator using one-dimensional CAs, one must check that the best affine approximation to the update rule(s) has a low probability of matching.

Matsumoto in 1998 showed how to construct CAs with maximal periods by making a small modification to Wolfram's rule 90 [MAT98]. Using rule 90 on cells with a null boundary to the left and a mirrored boundary to the right (such that $x_{m+1}(t) = x_m(t)$), he showed that there are 55 values of m (the number of cells in the CA) for $1 <= m <= 300$ such that the period is equal to $2^m - 1$. Matsumoto further shows how to extend this one-dimensional CA into a two-dimensional version. This result is significant because, previously, to construct a CA with a maximal period, one would be required to randomly

search for primitive polynomials, which is a difficult task.

Tomassini, Sipper, and Perrenoud in 2000 submitted a method for generating non-uniform two-dimensional CAs using genetic programming [TOM00a]. Starting with the best CA evolved from their technique, they extended it by hand to produce a new CA that out-performed all previously known two-dimensional CAs. Tomassini, et al. experimentally studied cycle lengths of two-dimensional CAs and presented empirical evidence suggesting that they increase exponentially with respect to the number of cells. Furthermore, they note that their two-dimensional CA does not require any time-spacing in order to achieve good randomness, whereas all previous one-dimensional CAs show lowered performance unless produced bits are excluded from the output stream.

Shackleford, et al., proposed in 2002 a new way to implement random number generators in one-, two-, and three-dimensional neighborhood-of-four CAs using field programmable gate arrays (FPGAs) [SHA02]. Their designs are built around four-input lookup tables with a one-bit register to efficiently exploit the nature of FPGAs. In their physical design experiments, they predict that this results in a maximum clock frequency between 214MHz and 230MHz. Furthermore, they show that their proposed CAs pass all of Marsaglia's DIEHARD [MAR98] tests, as they include all the raw p-values for the reader to verify. Shackleford, et al, also provide experimental results suggesting a linear relationship between the number of CA cells and the cycle length for their designs.

Seredynski, Bouvry, and Zomaya in 2003 used an evolutionary technique called cellular programming to select and find a set of one-dimensional CA rules that outperform all previously-known one-dimensional CA-based PRNGs in terms of quality of randomness [SER04]. They discovered a set of eight rules (seven 1-radius rules in addition to one 2-radius rule) that, when combined together, provide a stream of bits that appears more random than any of those rules used individually; this surprising result shows that strong PRNGs can be constructed from several weaker ones combined.

D'Antonio and Delzanno (2004) presented a method for encoding CAs into propositional logic so that they can be processed by SAT solvers [DAN04]. They detail a modular system that allows various queries to be constructed, most notably inverse reachability, which would allow the computation of an initial state (key) given the bit stream output. They show results from using their encoding methods to solve the boolean CA expressions and the associated execution times using the zChaff [MOS01] SAT solver. Their system allows one to test small versions of CAs to extrapolate practical strength as well as compare CA designs against one another.

Oliveira, Coelho, and Monteiro (2004) examine CAs with bi-directional toggle rules in [OLI04] and show that they achieve much more diffusion of plaintext bits into the ciphertext as compared to the original proposal by Gutowitz in [GUT93]. Left toggle rules are those such as rule 60 that only take into account the left and center bits of the

neighborhood (rule 60 simply calculates the exclusive-OR of the left and center bits); rule 30 is bi-directional in that it accounts for both left and right neighbors. The authors empirically show that when using bi-directional toggle rules, a single bit change in the plaintext will cause half of the ciphertext bits to change. Furthermore, Oliveira, et al. show that their method provides immunity from differential cryptanalysis. However, while the original system had a key space of $2^{2^{V-1}}$, using bi-directional toggle rules lowers the key space to $2^{2^{V-2}}$ (where V is the neighborhood size).

Xeulong, Jiwen, Manwu, and Fengyu in 2004 suggested a new design for non-uniform one-dimensional CAs that incorporates a cell's value from a previous time step [XUE04]. That is, while the traditional CA's update function consists of: $s_i(t+1) = f(s_i\text{-}1(t),s_i(t),s_i+1(t))$ (where s is the state of the CA, $s_i(t)$ is the state of cell i at time t, and f is the evolve function of the CA), their proposed "extended" CA design is defined as: $s_i(t+1) = f_i(s_i^{t-1}, s_{i-1}^{t}, s_i^{t}, s_{i+1}^{t})$. This represents a new technique that has never been used in the design of CAs.

In 2005, Popovici and Popovici generalized Wolfram's rule 30 to use an arbitrary number of states (powers of two) to create a reversible CA [POP05]. With this scheme, they demonstrate how to construct a new cryptosystem using ideas from [LAF96], though it lacks any kind of rigorous security analysis. The authors only show how the encryption and decryption methods work, but do not show any results from randomness testing, In

the conclusion, they merely state, “hopefully [this scheme] provides better security.” [POP05]

Xuelong, Qianmu, Manwu, and Fengyu (2005) built upon their previous efforts in [XUE04] along with genetic programming introduced by Tomassini et al. (in [TOM00a] and [TOM01]) [XUE05]. They show that genetic techniques can locate extended CAs that are suitable for cryptography, as the results from the Diehard tests ([MAR98]) are presented. Unfortunately, their results are difficult to independently verify, as the specific CAs they discovered genetically are not mentioned, nor are any raw data from the statistical tests given.

Alvarez and Li (2006) present 17 guidelines for proposing new cryptosystems based on chaotic systems such as cellular automata [ALV06]. In surveying published research, the authors note that many proposed systems lack concrete details necessary for cryptanalysis, and/or do not present enough information to show its viability in terms of cost and efficiency. Alvarez and Li state that implementation performance such as software run-time are necessities that are critical in demonstrating usefulness of the cryptosystem. Their paper is useful for designers of new CA-based cryptosystems.

Delgado, Vidal, and Hernandez in 2006 extended Tomassini's cellular programming technique in [DEL06] to include statistical tests to the fitness function in addition to the entropy measurement already used; this helps eliminate unfit rules more

quickly. This extended technique was used to examine non-uniform one-dimensional CAs between one and five arbitrary of neighbors. They discovered a rule (rule 2572018122) that appeared to perform as well as, or better than, well-known pseudorandom number generators that are actively used, such as ANSI X9.17, Blum Blum Shub, and G-DES. Significantly, they suggested that future CA designs can perform well using non-localized neighbors.

Mukhopadhyay and RoyChowdhury (2007) study complemented group CAs (those that contain XNOR logic, such as rule 153) and lay groundwork that is useful for building cryptosystems upon them [MUK07]. Using matrix algebra, the authors present proofs for calculating cycle lengths of CAs, which is a critical component for certain cryptographic applications (as unknown or short cycle lengths could be a major problem for a cryptosystem). The authors claim that the results of their paper would assist designers of block ciphers and key agreement protocols since reversibility can be ensured by utilizing half of a CA's known cycle length for diffusion functions; this would ensure that progressing an additional half-cycle length results in the initial state. Mukhopadhyay and RoyChowdhury suggest that the strengthening of CA primitives would help their acceptance into mainstream cryptography.

As we have seen, there has been significant interest in genetic programming for discovering configurations of CAs in situations that manual searching cannot be done

(i.e.: in the case of large neighborhood sizes). These techniques have produced good results, and are likely to remain viable methods for studying CAs in the future.

Techniques for ensuring maximal period and maximal bit diffusion are critical components for cryptographic systems. The papers presenting methods for ensuring those properties are certainly invaluable for system designers.

3. BLOCK CIPHERS

In this chapter, we report on the literature found regarding the construction of block ciphers with cellular automata.

Srisuchinwong, York, and Tsalides (1995) introduce a symmetric block cipher based on the novel idea of autonomous and non-autonomous CAs [SRI95]. Autonomous CAs are those whose state transitions are not dependent upon external input; non-autonomous CAs mix in external input while computing a cell's next state. The authors use an autonomous CA to compute session keys. The plaintext is used as an initial state of a non-autonomous rule-218 CA and evolved, taking inputs from the session keys into consideration. Their resulting design gives a cipher that operates on 16-bits of plaintext at a time and uses a 96-bit key. While the authors do suggest that a high rate of encryption is possible (320Mbps on hardware clocked at a modest 20MHz), they unfortunately do not present any security analysis in terms of linear cryptanalysis nor differential cryptanalysis.

In 1997, Lafe studied how to use cellular automata transforms to compress images as well using them to construct a block cipher [LAF96]. He suggests that transforms exist which provide the avalanche effect, though he offers no evidence nor analysis. Very little security analysis is presented; no discussion on linear cryptanalysis nor differential

cryptanalysis is included. In this paper, Lafe extends ideas from data compression into the realm of security and suggests an avenue that can be further explored by future research.

Shaw, Chatterji, Maji, Sen, Roy, and Chaudhuri (2001) propose a new technique called "encompression," consisting of data encryption and compression using CAs [SHA01]. Specifically, the authors design an algorithm that is optimized for processing video of human portraits. To compress the video, a technique called "vector quantization" (or VQ) is used, which is claimed to be more efficient using a cellular automata implementation. Encryption is done using the same technique as in [SEN03]. The authors show that real-time video compression and encryption can both be efficiently realized using cellular automata in hardware.

Sen, Shaw, Chowdhuri, Ganguly, and Chaudhuri in 2002 [SEN02] presented a new 128-bit cellular automata-based block cipher using a combination of linear transformations, an affine CA operation (using $GF(2^8)$), and a non-linear, non-affine CA operation (also using $GF(2^8)$) to add security and overcome the problems encountered in [NAN94]. The authors explain the security implications of each of the major components; the initial linear transformation is noted to introduce a degree of randomness to the input, the affine CA is noted to produce a very large set of outputs (2^{128}), and that the final key mixing step makes the block "totally unpredictable." Sen, et al., empirically

show that the block cipher is immune to differential cryptanalysis and also empirically show that it surpasses both DES [NIS99] and AES [NIS01c] with respect to Shannon's security quotient [SHA49]. Compared to the execution time of AES, the cipher is shown to be approximately five times faster in software, and approximately eighteen times faster in hardware.

Zhang and Li (2002) propose a hardware-optimized block cipher based on a two-dimensional additive CA design [ZHA02]. Whereas the block cipher proposed in [NAN94] uses n x m operations for encryption and decryption, the proposed cipher uses $n + m - 1$ operations (where n is half of the CA group order and m is the number of message blocks). The authors suggest that their design could be used for public key cryptography when using Galois fields, though they do not show how this is possible. Furthermore, the authors do not include a thorough security analysis of their cipher with respect to linear and differential cryptanalysis; only the large key space is shown. Because the authors reference the work done in [NAN94] several times, the reader would naturally wonder if the attacks published on [NAN94] are applicable to this new design. Unfortunately, Zhang and Li do not present any evidence showing that their cipher is immune to those problems.

Sen, Hossain, Islam, Chowdhuri, and Chaudhuri (2003) propose a block cipher very similar to the one in [SEN02] with the only notable change being a modification of

the non-affine transformation step (the 128-bit non-linear group CA was replaced with a "non-linear reversible Control Major Not gate.") [SEN03]. In this paper, the authors present the cipher design as an efficient solution for embedded applications; it is noted that the optimized software version of AES uses about 65 kilobytes of memory space, whereas the proposed cipher uses only 3.8 kilobytes.

Srebrny and Such (2003) propose a block cipher using two-dimensional CAs [SRE03]. Their design uses three automata (two of which are irreversible), along with geometric computations to help with the diffusion of bits. The first phase of encryption of a 256-bit block of plaintext involves setting the plaintext as the initial state of the main 32x8 CA, and using the symmetric key to configure the initial state and rules of the 2x8 and 32x8 auxiliary CAs; the evolution of the plaintext is dependent upon data extracted from the evolved auxiliary CAs. Next, the main CA is diffused further using toggle rules, and finally is mixed using transformations on 2x2 subsections. The authors provide sufficient details on how their algorithm works, as well as an implementation that encrypts Microsoft Word documents, though unfortunately, they do not present any security analysis of their cipher; neither linear nor differential cryptanalysis are provided.

In 2003 and 2004, Bao showed that the block cipher proposed in [SEN02] was insecure against a chosen plaintext attack [BAO03][BAO04]. Using only a couple hundred chosen plaintexts, an efficient algorithm is presented that reconstructs the

encryption and decryption transformation functions, resulting in a significant breakdown of the cryptosystem. Bao also shows that minor changes to the design (such as adding a permutation on the plaintext bits, changing the affine CA into a non-affine CA, and/or other changes) does not improve the security significantly; a complete re-design would be needed to defend against the vulnerability.

Mukhopadhyay and Chowdhury (2004) propose a 128-bit block cipher design based on CAs that incorporates two non-linear transformations in such a way that the cyclical nature of the linear transformation is not disturbed [MUK04]. Using rule 153, the authors show that a cyclical CA can be built, as its cycle length grows roughly in-step with the number of cells; this implements the linear part of the cipher. The non-linear parts of the cipher are simply Wolfram's rule 30. Unfortunately, the authors do not present any security analysis whatsoever.

Seredynski and Bouvry in 2004 designed a symmetric block cipher using reversible CAs [SER05]. Specifically, their cipher uses a 224-bit key to operate on 64-bit blocks of plaintext over 16 rounds. Reversible CA s are pairs of CA rules that complement each other such that evolving a state *n* times with one rule, then evolving the resulting state another *n* times with its complementary rule will result in the original state (thus it is termed 'reversible'). Seredynksi and Bouvry present a full block cipher design in this paper detailing all aspects of operation along with experimental results showing

that the avalanche effect applies (i.e.: that a change in one bit of plaintext or key will result in a change of at least 50% of the output bits). The authors present a short cryptanalysis section discussing the presumed difficulty of a brute-force attack on the system.

Del Rey proposed in 2005 an entire cryptosystem based on CA [REY05]. His design includes a new concept termed "reversible memory cellular automata" that contains two properties: 1.) that the CA operations are reversible (i.e.: reverse-evolution is possible to obtain the initial state given an evolved state), and 2.) that the CA evolution function takes several previous time-steps into consideration (four previous time states are used in his actual algorithm), similarly to the design suggested by Xeulong in 2004. Del Rey shows that the suggested block cipher is secure against brute-force, ciphertext-only, known-plaintext, and chosen-plaintext attacks.

Liu, Cheng, and Wang (2005) present in [LIU05] another attack on the block cipher proposed by Sen, Shaw, Chowdhuri, Ganguly, and Chaudhuri ([SEN02]). The best known attack (by Bao in [BAO03],[BAO04]) showed that equivalent encryption and decryption functions could be constructed given a couple hundred chosen-plaintexts along with small additional computations. In this paper, the authors show a more efficient attack that requires only two chosen plaintexts and two quick computations that results in a complete break in the system; several internal mechanisms that could not be

previously discovered using Bao's attack can be computed, along with recovery of the key.

Joshi, Mukhopadhyay, and RoyChowdhury proposed a block cipher in 2006 [JOS06] that built upon previous work in [MUK04]. They presented a novel way of constructing a non-linear S-box using a non-linear CA that is highly resistant to differential cryptanalysis. Furthermore, their design includes a separate "diffusion-box" (D-box) mechanism consisting of linear CA that is responsible for achieving the Avalanche criterion. Key mixing is implemented using addition and subtraction in $GF(2^n)$ (where n is the block size) so as to prevent linear cryptanalysis. The authors give empirical evidence in their security analysis of the cipher that suggest high resistance to linear and differential cryptanalysis as well as satisfaction of the Avalanche criterion. According to Joshi, et al., this paper is the first to present a fully-detailed CA-based cipher specifying all components necessary to allow its cryptanalysis, such as block size, key size, key scheduling mechanism, etc.

Marconi and Chopard (2006) show how to construct a topology for a CA-based block cipher called "Crystal" [MAR06]. It is suggested that two-dimensional cellular automata can be used to implement the generic functions that are specified in the algorithm, leading to a cipher design that is pluggable in nature. The authors present evidence that shows that the algorithm becomes increasingly more difficult to

cryptanalyze given additional rounds of encryption. Unfortunately, the authors do not fully specify how to construct a concrete system or how to apply two-dimensional CAs to the Crystal algorithm. No analysis is done regarding differential or linear cryptanalysis.

Sung, Hong, and Hong (2007) show that the block cipher proposed by Joshi, et al. in [JOS06] is insecure due to its conjugate property [SUN07]. It is shown that any plaintext can be easily decrypted using a chosen-plaintext attack; by encrypting a ciphertext $2n$-1 times (where n is the block size), the conjugate property in combination with the involutional design results in the original plaintext.

As it has been seen, many block cipher proposals are not fully specified in terms of key size, key scheduling, or defining certain functions within the cipher. As noted by Bao in [BAO03] and [BAO04], this makes cryptanalysis much more difficult, if not impossible (though he was still successful in cryptanalyzing the cipher by Sen, et al. in [SEN02]). These unspecified ciphers are perplexing because they have no hope of gaining acceptance by the cryptographic community since they cannot be implemented and/or scrutinized, yet they have been published anyway.

Of the papers that have been fully specified, many of them have been broken. And many of the published proposals do not sufficiently present evidence that they are resistant to common attacks, such as linear and differential cryptanalysis. These, too, are perplexing, as any respectable cipher design is not considered complete without

supporting evidence of its viability.

While it is apparent that block ciphers based on CAs are not yet mature enough to be used in a production setting, it should not be considered the fault of the CA primitives; the weaknesses seen in this chapter are not the result of the CAs themselves, but are rather the result of general difficulty in designing cryptographic systems.

4. STREAM CIPHERS

As noted before, stream ciphers with CAs were first proposed by Wolfram in [WOL86]. This is perhaps the most natural fit for CAs in cryptography, due to the fact that they can generate a stream of seemingly-independent random bits. The bits can form a key-stream that can be combined easily to the plaintext with the XOR operation to produce ciphertext, and later decrypted by XOR'ing the same key-stream to the ciphertext to recover the plaintext. In this way, the stream cipher contains a design similar to that of the provably-secure one-time pad (OTP). Though the OTP requires a truly random source while CAs are pseudo-random, in theory a CA-based stream cipher can have an upper-bound security value dependent upon its cycle length (which in some cases is 2^n, where n is the number of cells). An upper-bound value in security helps set a goal that designers can attempt to achieve.

Unfortunately, stream ciphers have been passed over in favor of block ciphers. No stream cipher has been standardized by the U.S. government, unlike the block ciphers DES [NIS99] and AES [NIS01c] As a result, stream ciphers have not been subjected to the same rigorous public review that block ciphers have enjoyed, with the exception of the RC4 stream cipher (which, strangely enough, has never been officially published, though an unofficial description is available in [ANO94]) that is used in practical systems such as

Transport Layer Security [DIE06] and Wi-fi Protected Access [IEE04].

Regardless, stream ciphers are still of interest within academic circles, and can be practical in implementing a defense-in-depth strategy. In the unlikely, but highly disruptive event that a standardized block cipher is broken, a well-reviewed stream cipher can be a viable alternative. For that reason, we are drawn to the literature of CA-based stream ciphers.

Meier and Staffelbach (1991) demonstrated an attack on stream ciphers built using rule 30 [MEI91]. Their known-plaintext attack exploits the patterns in the bit sequence adjacent to the key-stream that are generated by rule 30 to more effectively reconstruct the initial CA state. They empirically show that the effective key size of a 200-bit key is 14.5 bits, with respect to a 50% chance of successfully recovering that key. Extrapolating their results, they suggest that a key size approximately 750 to 900 bits long has an effective size of only 50 bits; this is much smaller than the key space of 2^{750} to 2^{900} previously thought those keys would occupy.

Nandi, Kar, Chaudhuri in 1994 presented three new CA designs: one for a block cipher, and two for stream ciphers [NAN94]. Their novel block cipher design revolves around the use of rules 51, 153, and 195 to generate an alternating group to construct a set of fundamental transformations. These transformations are used in conjunction with a programmable CA (PCA) design, which relies on a separate control structure to select the

active rule configuration. Nandi, et al., show that it is resistant to ciphertext-only attacks as well as known and chosen plaintext attacks. Their first stream cipher is described as a PCA with ROM; the read-only memory (ROM) holds a sequence of rules to program the CA structure with; it is designed in such a way to be highly efficient in terms of VLSI design. Their second stream cipher is described as a two-stage PCA, consisting of a controlling CA structure whose output programs another CA's active rule; both are suggested by the authors to either be uniform, or hybrid configured with rules 90 and 150. While no empirical evidence is presented regarding the randomness of the key-stream, the authors do show how their stream cipher designs are resistant to known ciphertext and known plaintext attacks.

Blackburn, Murphy, and Paterson in 1997 showed that the block cipher and stream ciphers presented by Nandi, et al., in [NAN94] are weak [BLA97]. They point out that the block cipher is constructed entirely of affine transformations, which implies that the block cipher itself is affine. The first stream cipher (i.e.: PCA with ROM) is shown to be insecure due to its low linear complexity; the key-stream can be determined in complexity $O(L^2 l)$, where L is the number of cells in the PCA and l is the number of rules in the ROM. Lastly, Blackburn, et al., present a method to recover the key used in the two-stage PCA stream cipher assuming a small amount of the key-stream is known in time $O(2^{L+2})$, compared to $O(2^{3L})$ suggested by Nandi, et al. A further observation is

presented that suggests that this cipher is of linear complexity.

Mihaljevic showed in 1997 ([MIH97b]) a ciphertext-only attack against the PCA with ROM cipher presented in [NAN94]. By exploiting the linear structure of the cipher following the use of rules 90 and 150, he showed that the effective strength of the secret key of length $L(l + 1)$ (where L is the number of cells in the PCA and l is the number of rules in the ROM) is upper-bounded by $\log_2(lL + 2^L - 1)$. Mihaljevic presents an efficient algorithm to deduce the secret key given only ciphertext.

Mihaljevic (1997) gives an improvement in [MIH97a] to the PCA with ROM stream cipher design originally proposed in [NAN94]. By adding an unspecified time-varying permutation to the bit stream output, the generator is shown to be resistant to the attack in [MIH97b]. However, it is noted that trade-off to this security improvement is a speed slowdown by a factor of *n (*where *n* is the number of cells in the CA). Mihaljevic also shows a novel cryptanalytic algorithm with attempts to reconstruct non-overlapping parts of the initial state/secret key by analyzing parts of the output stream. Evidence is presented that shows that the new improvement is secure against this novel attack if the number of cells in the CA is greater than 120.

Mihaljevic, Zheng, and Imai in 1998 [MIH98b] extended the results in [CAT96] to construct a fast cryptographic key-stream generator. The authors show how to combine a linear CA over GF(q) with principles of general stream cipher design such as time variant

non-linear mapping/filtering and the shrinking principle in a novel way such that weaknesses in the individual designs are not present in the final result. The security of their proposal is examined, with suggestions that it is immune to all previously known attacks, as well as robustness against general problems such as low complexity (which would allow a simpler equivalent generator to be constructed), small period, or weak statistical properties (though note that the paper merely references other works and does not directly show the reader).

Tomassini and Perrenoud (2000)[TOM00b] suggested a design for a one-dimensional hybrid CA key-stream generator that is resistant to the attack by Meier and Staffelbach in [MEI91]. Their proposed design involves using the key to also program the CA's rule vector using rules 90, 105, 150, and 165 (shown to make a good PRNG in [TOM00a]). The authors prove that using the attack in [MEI91] on this method would require $2^{(5n-9)/2}$ attempts (where n is the number of cells in the CA) in order to recover the key. Furthermore, their analysis shows that expanding their design from one to two CA dimensions results in an even greater resistance to attack (though the increase is not explicitly specified).

Tomassini and Perrenoud (2001) extended their results in [TOM00b] by applying the same genetic programming algorithm in [TOM00a] to the problem of finding good one- and two-dimensional non-uniform generators that are suitable for cryptography

[TOM01]. Because there are many more possibilities for non-uniform two-dimensional CAs, the authors suggest that genetic programming is best suited for finding good pseudo-random generators. As in [TOM00b], the authors show that their discovered CAs are resistant to the attack presented by Meier and Staffelbach in [MEI91]. Unfortunately, the exact two-dimensional CA structures are never explicitly listed by the authors, and thus their results are not easily reproduced, nor verified.

Badr (2002) illustrated a cryptogram incorporating a stream cipher along with reversible CAs and CA transform functions [BAD02]. Unfortunately, no specific details are presented regarding how any of the CA functions are constructed, nor is any security analysis given.

Bouvry, Seredynski, and Zomaya (2003) extended the results in [TOM00b] and [TOM00a] by applying the same genetic programming algorithm in [TOM00a] to the problem of finding good one-dimensional non-uniform generators that are suitable for cryptography [BOU03],[SER03]. The authors discovered that the radius-1 rules 86, 90, 101,105, 150, 153, 165 along with the radius-2 rule 1436194405 generate random streams with high entropy and pass the chi-square test, serial correlation test, and the FIPS 140-2 tests [NIS01b]. The rules passed all 23 of the DIEHARD tests [MAR98], whereas the rules proposed in [TOM00b] pass only 11 of them. Furthermore, the authors show that their scheme has a higher key space ($8^N * 2^N$) compared to the design of [TOM00b] ($4^N *$

2^N).

Len, Encinas, Encinas, White, del Rey, Sanchez, and Ruiz (2003) study all of the radius-1, one-dimensional uniform automata (a.k.a. the Wolfram cellular automata) and their applicability to cryptography [LEN03]. Beginning with all 256 rules, bit streams are generated and evaluated against five statistical tests: the frequency test, the serial test, the runs test, the autocorrelation test, and the poker test. Of the 21 rules that passed the statistical tests, 12 of them were found to contain sufficient linear complexity. The authors then show a generic algorithm based on Meier and Staffelbach's attack in [MEI91] that works against the 12 remaining rules to recover the initial state/key. Because of this, it is concluded that Wolfram cellular automata are not suitable for use as stream ciphers.

Franti, Slav, Balan, and Dascalu (2004) present hardware schematics detailing the construction of a 4-rule programmable CA stream cipher that can be adapted and extended to implement a specific design [FRA04]. Their work can be used as a starting point to construct a custom hardware-based stream cipher.

Rubio, Encinas, White, del Rey, and Sanchez (2004) study the use of hybrid linear CAs for use as stream ciphers [RUB04]. Starting with all seven one-dimensional, radius of one rules (a.k.a. the Wolfram rule-type), the authors craft pairs of rules that pass basic pseudo-random tests, then submit the remaining eleven linear rule pairs to more stringent

tests. The eleven pairs are run multiple times through the serial test, poker test, run test, and autocorrelation test in periodic and null-boundary conditions. It was found that the (60,90), (90,150), and (150,240) pairs passed all tests.

Bouvry, Klein, and Seredynski (2005) show that the key-stream generator in [SER04] can generate patterns if keys with a certain pattern are used [BOU05]. They suggest that a working implementation of the generator should check the key to ensure that the pattern is not present, as any key that contains that pattern is a weak key. The authors also note that there is no proof that there are no more key patterns that produce the same result. It is suggested that an implementation check the entropy of the key itself before accepting it.

Szaban, Seredynski, and Bouvry (2006) re-examined the set of CA rules proposed in [SER04] since it was found in [BOU05] that "some specific assignments" could lead "to bad statistical quality" [SZA06a],[SZA06b]. Using a genetic algorithm, ten hybrid one-dimensional CA consisting of five radius-two rules were found to produce sequences of high entropy that also pass the FIPS 140-2 statistical tests [NIS01b]. Even though the DIEHARD tests [MAR98] were not used, nor any other statistical testing suite, the authors nevertheless claim that their discovered rule sets are suitable for use in a key-stream generator.

The papers centering on stream ciphers have demonstrated several key points that

future designers must acknowledge. The attack by Meier and Staffelbach in [MEI91] shows that information regarding neighboring cells is sensitive and can be exploited to find the key. Any future key-stream generator must ensure that this attack does not apply. Secondly, we now know that no Wolfram cellular automata (i.e.: one-dimensional, one-radius CAs) are suitable for cryptographic use from Len, Encinas, Encinas, White, del Rey, Sanchez, and Ruiz in [LEN03]. Lastly, we are made aware by Bouvry, Klein, and Seredynski in [BOU05] that key-stream generators based on CAs can contain weak keys, though a general method to deduce them is still not known.

It should be noted that most papers in this chapter do not directly show any results of randomness testing. Of the ones that do, the criteria which the tests are based upon are not given. This makes their claims of randomness less than convincing. Future designs must include both the results as well as the criteria upon which they were found.

A key observation relating to the novel work done in chapter 9 is that the hybrid design by Tomassini and Perrenoud in [TOM00b] seems to provide good protection from the attack by Meier and Staffelbach in [MEI91]. This point has influenced the process in designing the Sum-4 cipher described in chapter 9.

5. HASH FUNCTIONS

Several authors have suggested designs for constructing a hash function based on CAs, including the high-profile cryptographers Damgård and Daemen (separately). CAs seem to lend themselves nicely to hash function designs due to their regular nature and high parallelism.

Damgård (1990) proposes a way to construct a 128-bit output hash function using rule 30 over 512 cells [DAM90]. To ensure that collisions are difficult to compute, he proposes that the message is padded with a random constant 256-bit string, then used as the initial state. After iterating 256 generations, a stream of 128 bits is taken as the hash of the message block and XOR'ed together with the previous message block. Damgård presents a proof that the padding significantly hinders attempts in computing collisions, but an algorithm to do this was nevertheless discovered by Daemen, et al, in [DAE93].

Daemen, Govaerts, and Vandewalle (1993) in [DAE93] show how to find collisions with Damgård's CA-based hash function presented in [DAM90]. They also present their own 257-bit hash function called Cellhash that uses rules 30 and 150 to provide bit confusion and diffusion, respectively, along with a final permutation to ensure that every bit of each generation g is dependent upon every bit of generation g-3 in a non-linear way. Using ideas from differential cryptanalysis, the authors show that confusion is sufficiently

achieved, as small changes in the CA state results in many possible difference patterns, each with a very small probability after few rounds. Lastly, Daemen, et al., suggest that the Cellhash algorithm is highly efficient in hardware, as they conservatively estimate that 0.3 Gb/s is possible using only a 10MHz clock.

Mihaljevic, Zheng, and Imai (1998) propose a Davies-Meyer type hash function using programmable linear additive CAs [MIH98a]. Their design uses a CA to implement both the compression function as well as the output function. Interestingly, the output function is a variation of the key-stream generator proposed in [MIH97a]. An accompanying security analysis shows that the compression and output functions are immune to pre-image, second pre-image, and collision attacks, as well as other known attacks. The hash function is shown to be fast, with an efficient hardware implementation suggested. The authors note that their hash function is significantly faster than dedicated hash functions such as MD5, SHA-1, RIPEMD-160, and HAVAL.

Mihaljevic, Zheng, and Imai in 1998 extended the results in [CAT96] to construct a fast cryptographic iterative hash function using linear CA over GF(q) [MIH98c]. Their design uses a CA to implement both the compression function as well as the output function. An accompanying security analysis shows that they are immune to pre-image, second pre-image, and collision attacks. The hash function is shown to be fast, with an efficient hardware implementation suggested. Aside from their novel designs for the

compression and output functions, this work is notable for proposing a word oriented hash function using CAs instead of previously proposed bit oriented hashing.

The work done in the area of hash functions with CAs is of higher quality than in other domains. Authors proposing hash algorithms have provided full specifications as well as in-depth security analyses, paving the way for acceptance by the mainstream cryptographic community. While the U.S. government already has standardized hash algorithms, in recent times the National Institute of Standards and Technologies (NIST) has called for proposals for a new hashing standard to protect against new weaknesses found [NIS07]. It is entirely plausible that the work done in this field could be applied to the task and a proposal be filed for a new hashing standard based on CAs.

6. MESSAGE AUTHENTICATION CODES

Two papers have explored the use of CAs to generate message authentication codes (MACs). The general idea behind them is to use the plaintext as the initial state, then evolve the CA a certain number of steps to compute the MAC while mixing in key information.

Dasgupta, Chattopadhyay, and Sengupta (1999) propose a design for creating a message authentication code (MAC) using linear rules 90 and 150 [DAS99]. By arranging an n-bit CA in a two-predecessor single-attractor (TPSA) configuration (where each state as exactly two predecessors), a secret key is set to the initial state and n-bit blocks of the message are used to configure the rule vector for each time step. To increase complexity and to ensure that the resulting MAC is not all zeros, an additional complemented evolution is performed before loading the next n-bit message block. The final CA state is the MAC, which is dependent upon the key and the message. The authors show that, due to the TPSA design, 2^k possibilities for the initial state/key exist for any message and MAC pair (where k is the number of n-bit message blocks). However, the difficulty of modifying any parts of the message is not discussed nor analyzed. Thus, the security of their scheme is unknown.

Mukherjee, Ganguly, and Chaudhuri in 2002 [MUK02] show how to construct a message authentication code (MAC) using CAs over GF(2^p). They point out that their scheme is immune to brute force attacks since the message digest length and key length can be variable. It is also immune to the Extension attack (also called the Paddle attack), as well as a variation of the differential cryptanalysis attack (that would enable the discovery of the secret key through observation of the differences in digest pairs). Mukherjee, et al., further show that their scheme can be used to construct sensitive watermarks into digital images. However, the system has been successfully cryptanalyzed by Lee, Hong, and Kim in 2006. Lee, et al. showed that Mukherjee's (et al) system is vulnerable to a chosen-message attack due to its inherently regular design [LEE06].

While the idea of constructing a MAC with CAs seems promising, the fact that only two papers have been published on the topic (one of which lacks the necessary security analysis, and the other which has been broken), clearly shows that it is not headed towards any practical result. Nevertheless, this area seems ripe for future research, as CAs are known for their efficiency in parallel hardware which is attractive for MACs.

7. PUBLIC KEY CRYPTOGRAPHY

The application of CAs to public-key cryptography is an interesting idea. Unfortunately, only one paper on the topic has been published in over twenty years, suggesting that it is largely an academic or abstract idea.

Guan, in 1987, proposed the first-ever method of designing a public-key cryptosystem with cellular automata [GUA87]. He suggests that by combining a set of multi-fold linear invertible rules together, the resulting function is no longer partially linear. This fact would allow a public key to be made out of the combination, as the private key (the individual functions of the rule) would be very difficult to compute. Guan presents a small example illustrating the concept, but does not give any rigorous security analysis, nor does he show how encoding for arbitrary-sized messages can be efficiently accomplished. Furthermore, no method to efficiently generate public keys is given.

8. GRAPHICS CRYPTOGRAPHY

A small group of researchers have published several CA-based cryptosystems that are specifically crafted for digital images. As two-dimensional CAs can be easily constructed to exactly match a two-dimensional image in length and width, they can be naturally combined in order to achieve diffusion, confusion, or authentication.

Maranon, Encinas, Encinas, del Rey, and Sanchez (2003) propose a cryptosystem for graphical color images using two-dimensional reversible CAs [MAR03b]. In their system, the image is used as the initial seed for the two-dimensional CA, and the Blum Blub Shub (BBS) pseudo-random generator is used as a key-stream. A radius of two around each pixel, not including that pixel (resulting in a neighborhood size of 24) is taken into account by the updating function along with the BBS stream in order to encrypt that pixel. Only one generation is computed to encrypt the image. The authors note that no expansion of the plaintext occurs and that no loss of information happens, unlike previous graphics cryptography designs. However, the authors do not present any security analysis whatsoever as to the viability of their design to resist attacks. Surprisingly, the authors present an image and its associated ciphertext which seems to contradict their assertion that their algorithm is useful for cryptography; the ciphertext appears to contain visible artifacts left over from the plaintext.

Maranon, Encinas, and del Rey (2005) introduce a (n, n)-threshold secret image sharing scheme (where an image is split into n tokens among n parties such that all n parties must agree to contribute their token in order to reconstruct the original image) using reversible two dimensional CAs [MAR05]. The original image is loaded as the CA's initial state, and is evolved using linear rules that incorporate previous time steps (other than the immediately previous state; these are termed linear memory cellular automata, or LMCA). The trusted share distributor will compute the tokens by evolving the initial state (original image) using different transition functions over different parts of the original image. The original image is thus recovered by reversing the evolution of the tokens and combining them all to recover the image. The authors show that this cannot be done unless all n tokens are combined, otherwise no information is leaked (i.e.: that it is a perfect scheme). It is also demonstrated that their scheme is resistant to statistical analysis in that the tokens are uniformly distributed.

Encinas, del Rey, and Sanchez (2005) propose an image authentication protocol using two dimensional reversible CAs [ENC05]. Specifically, their system will use a random image as the initial state, then set the first and second states to the first and second halves of the image, respectively. A transition function that incorporates the previous three time states is run in order to generate a MAC that is half the size of the image. The authors provide a detailed security analysis, showing that their scheme satisfies the ava-

lanche effect, that the generated fingerprints have a uniform distribution, and that it is collision resistant as well as being pre-image and second pre-image resistant. While the size of the MAC is linear with respect to the source image, the authors note that this may be changed to a constant size with future research.

Cryptographic functions that operate only on images are interesting from the academic perspective, but are of little practical use in the real world. The need for special-purpose encryption on images is easily dwarfed by the need for general-use encryption that can operate on arbitrary data such as information in a database, network protocols, or on general data files. Nevertheless, the work done in this field might one day be applied in mainstream methods.

9. A NOVEL CIPHER DESIGN: THE SUM-4 CIPHER

9.1 Influence and Design

Now that we know what work has been published regarding cryptography with cellular automata, we present a novel stream cipher design that resists all known attacks in the literature.

While performing the literature search, our attention was drawn to the attack on a stream cipher based on rule 30 described in [MEI91] by Meier and Staffelbach in 1991. This attack depends upon the fact that a single rule is in effect for all generations and that rule 30 is partially linear, which leaks some information about the neighboring cells. From there, an attacker can use information from the neighboring cells and key-stream to deduce information about the key. As a result, the key size must be increased greatly in order to be secure against the attack.

Tomassini and Perrenoud suggested in 2000 that a hybrid stream cipher can be constructed by assigning rules 90, 105, 150, or 165 to each of the cells, where the initial state and rule assignment both consist of the key [TOM00b]. They show that this thwarts the attack in [MEI91].

We propose an idea similar to [TOM00b]. However, instead of assigning each cell

a static rule at setup time, we propose that all cells use a single rule that is changed each generation based on control bits. In order to preserve the balance of 0 and 1 bits in the state of the CA, we choose to use rules that have an equal number of 0 and 1 outputs. Collectively, we term these rules the *sum-4* rules.

Figures 9.1, 9.2, and 9.3 show sample sum-4 rules. The black squares represent cells with a value of 1; white squares represent cells with a value of 0. The row of three squares represents the rule inputs, and the square below them is the rule's output:

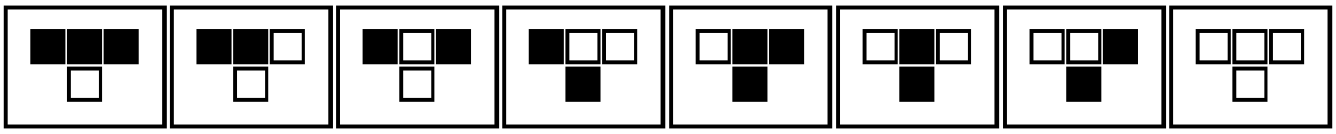

Figure 9.1. Rule 30's inputs and outputs.

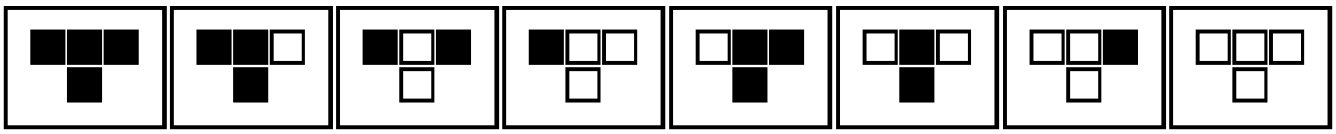

Figure 9.2. Rule 204's inputs and outputs.

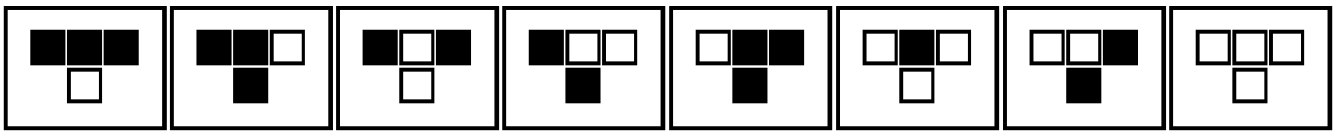

Figure 9.3. Rule 90's inputs and outputs Notice that its outputs are symmetrical.

It was found through exhaustive search that there exist 70 sum-4 rules. Of them, 6 are symmetrical in that reversing their outputs results in the same rule (such as rule 90). Removing the symmetrical rules, we are left with 64 sum-4 rules. They are: {15, 23, 27, 29, 30, 39, 43, 45, 46, 51, 53, 54, 57, 58, 71, 75, 77, 78, 83, 85, 86, 89, 92, 99, 101, 105, 106, 108, 113, 114, 116, 120, 135, 139, 141, 142, 147, 149, 150, 154, 156, 163, 166, 169, 170, 172, 177, 178, 180, 184, 197, 198, 201, 202, 204, 209, 210, 212, 216, 225, 226, 228, 232, 240}. A one-dimensional, cyclical-boundary, radius-1 control CA using rule 30 can be used to generate six random bits each generation by sampling six evenly-spaced cells. These bits are given to the rule-controller, which selects between the 64 sum-4 rules. The currently active rule is then applied to the cells of the main CA, which is another one-dimensional, cyclical-boundary, radius-1 CA containing the same number of cells. The first cell in the main CA each generation is sampled into the key-stream output. An XOR operation between the plaintext bits and the key-stream results in ciphertext; XOR'ing the ciphertext with the key-stream again results in the original plaintext (this is called a Vernam cipher [VER26]).

The table that translates bits from the control CA into the active rule number for the main CA is below:

Control CA Output	Main CA Rule	Control CA Output	Main CA Rule	Control CA Output	Main CA Rule	Control CA Output	Main CA Rule
000000	15	010000	46	100000	77	110000	105
000001	240	010001	116	100001	178	110001	150
000010	23	010010	51	100010	78	110010	113
000011	232	010011	204	100011	114	110011	142
000100	27	010100	53	100100	83	110100	135
000101	216	010101	172	100101	202	110101	225
000110	29	010110	54	100110	85	110110	139
000111	184	010111	108	100111	170	110111	209
001000	30	011000	57	101000	86	111000	141
001001	120	011001	156	101001	106	111001	177
001010	39	011010	58	101010	89	111010	147
001011	228	011011	92	101011	154	111011	201
001100	43	011100	71	101100	99	111100	149
001101	212	011101	226	101101	198	111101	169
001110	45	011110	75	101110	101	111110	163
001111	180	011111	210	101111	166	111111	197

Figure 9.4. The control CA bit output to active main CA rule lookup table.

In figure 9.4, notice that the sum-4 rules are not in ascending order. In the early stages of designing the cipher, each rule was paired up with another rule whose output is the reverse of the first. For example, rule 30's output is 00011110, and its reverse is 01111000, which is rule 120. These pairings were unintentionally preserved throughout testing, and while this ordering is not believed to have an effect on the properties of the

key-stream generator, we report them here in case this belief is incorrect and for the purpose of completeness.

The table in figure 9.4 is used to implement the *sum4* array in figure 9.5:

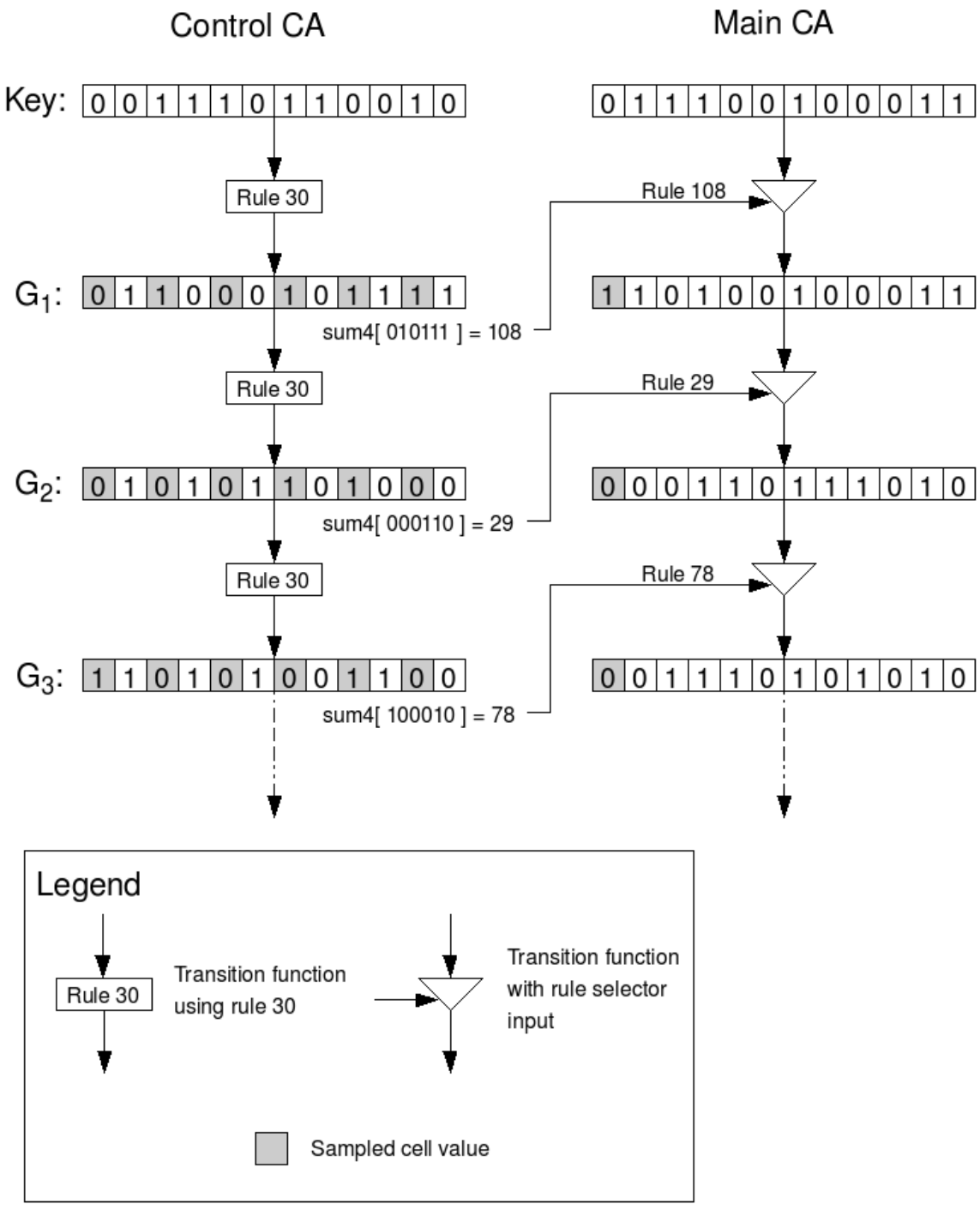

Figure 9.5. The Sum-4 cipher with 12 cells.

Figure 9.5 shows the Sum-4 key-stream generator with 12 cells. The control and main CAs are initialized with a 24-bit key (001110110010011100100011). The first generation is computed by transitioning the control CA using rule 30. Six equally-spaced cells are sampled (shaded in gray) to create an index into a table of the 64 sum-4 rules (as described in figure 9.4). The selected rule is made the active rule for the main CA, which is used to transition the key into the first generation. The first cell of the main CA (shaded in gray) is sampled and added into the key-stream output. Likewise, the second and third generations are computed, resulting in key-stream output of 100.

This design is more secure than [TOM00b] against the attack described in [MEI91] because an attacker that recovers part of the key-stream does not know what rules created the bits. Without this knowledge, an attacker has a harder time deducing the values of the neighboring cells.

It is hypothesized that this design could be augmented slightly to allow multiple cells per generation to be sampled without sacrificing security. This would allow multiple key-streams to be generated from roughly the same amount of computation or multiply the throughput of the cipher. As this hypothesis could not be tested due to time constraints, it is suggested as a course of future work in chapter 10.

Henceforth, this design shall be referred to as the Sum-4 cipher.

9.2 Testing Environment

All testing was performed on a 32-bit Intel Core 2 Duo T5500 CPU clocked at 1.66GHz with 2 GB of RAM on the Ubuntu Linux 7.10 operating system (kernel version 2.6.22-14). All computer programs are single-threaded and run on one CPU core only. The C compiler used is gcc version 4.1.3 20070929. The Java runtime environment and compiler are both IcedTea build 1.7.0-b21.

9.3 Randomness Testing

A key-stream must be random in order to be secure. If an attacker can predict the future output of the key-stream, then they can decrypt ciphertext, resulting in security failure. Ideally, an attacker must not be able to accurately predict even one bit in the entire key-stream.

When we initially approached the issue of randomness testing, it was not clear what software program to use. The literature most often used Marsaglia's DIEHARD test suite [MAR98], though there also exists Brown's DieHarder suite (which is a re-

implementation of Marsaglia's version under the GNU Public License [FSF91] along with extra tests), and NIST's Statistical Testing Suite (STS) [NIS01a]. Thus, the first step was to run all three suites on the key-stream output from the Sum-4 cipher and compare the results.

NIST's STS was hypothesized to be the best suite, as NIST should be able to marshal professional resources into publishing high-quality results. We were very surprised to find out that the STS was very poorly constructed. The implementation suffered from multiple critical problems. The documentation does not help the user understand how to invoke the program, nor understand what its options mean. It was found that the UNIX version often crashes or produces "underflow errors." The Microsoft Windows version does not even load due to missing DLL files under Windows XP. After these DLL files were later installed from public, untrusted sources, it was found that the summary of the test results were not properly listed, as individual tests would produce a result, but they would not be reported upon later (though the fact that untrusted DLL files were used could be a factor—however, this was the only way to make the program even functional). The Windows version could not be re-compiled from source code due to syntax errors. These problems affected all versions of STS, ranging from the initial 1.5 version to the latest version 1.8. In effect, the STS was found to be useless in randomness testing.

Next, Brown's DieHarder suite was considered. It was markedly better in quality than STS, though it revealed a few issues that brought into question its viability. Several tests are marked as unstable, suggesting that the suite is more of beta-quality than production-quality. As compared to Marsaglia's version, Brown's version took significantly more time to analyze the same input—about 7 times longer. Upon closer inspection, it was found that this was due to the way DieHarder re-tests the same blocks of bytes many more times, then applied multiple layers of statistics to them. Effectively, it gathered statistics on statistics, then reported the end result based on the meta-statistics. This explained why the same re-implemented tests in DieHarder were reporting many more failures versus the original DIEHARD. Lastly, DieHarder's "Count-the-1's" test was entirely incorrect; it reported a failure because the Sum-4's output supposedly had many more 1-bits than 0-bits. DIEHARD reported no such problem, and we created a custom program to verify that the output was indeed balanced. This outcome strongly suggested that DieHarder was not a good choice for randomness testing.

Thus, DIEHARD remained as the only viable randomness testing suite available at our disposal. However, no actual published test results from it could be found. Few papers surveyed in chapters 2 through 8 mentioned doing randomness testing at all. Of the ones that did, DIEHARD was used, though none of them presented the raw results, nor compared test results to those from a generator with known randomness properties.

This meant that the output of DIEHARD in and of itself could not yield definitive information; we would need to compare DIEHARD's results for the Sum-4 cipher against its results for generators with known good output. Only then could we determine if the Sum-4 cipher's output was random enough to be used in cryptography.

It should be noted that the DIEHARD suite outputs p-values for each of its tests. P-values are derived from test statistics that can be used to reject the null hypothesis (which, in this case, is that the generator outputs are random). The p-value is the probability that the test statistic would have a value greater than or equal to the observed value if the null hypothesis is true.

We found two published, non-cryptographic generators (also by Marsaglia) that are claimed to have good randomness properties: the Multiple With Carry (MWC) generator [MAR03a] and the Xorshift generator [MAR03a]. For each generator (along with the Sum-4 cipher), twenty 257-megabyte files of random numbers were created. Each file was analyzed using 14 of the 17 tests in the DIEHARD suite (three of them—the Tough Birthday Test, Up Down Runs Test, and OPERM5—were skipped because the meaning of their results could not be determined). The number of test failures were tabulated by generator (the less number of failures, the better the randomness). Unlike any published paper reviewed in chapters 2 through 8, we specify the criteria for passing a randomness test: if more than 10% of the two-tailed p-values resulting from a single test

(as some tests produce over 20 raw p-values) are greater than 0.975 or less than 0.025, then the test has failed at the 5% level. Otherwise, the test has passed. Similarly, for one-tailed p-values, if more than 10% of them are less than 0.05, then the test has also failed at the 5% level; otherwise the test has passed.

As DIEHARD merely presents the user many raw statistics, we edited its source code to output test success or failure based on the criteria specified above. Strangely, it was found during this time that no obvious distinction was made either in the program output or the source code whether the p-values reported were one- or two-tailed. It seemed as though Marsaglia assumed that the user would be an experienced statistician that would intuitively sort out the nature of the p-values in the sea of output. Thus, research was conducted to determine the type of p-values reported so that the pass/fail criteria could be properly applied. This revealed that two-tailed p-values were used in 14 of the 17 tests; unfortunately, the nature of the p-values from 3 tests (the Tough Birthday Test, Up Down Runs Test, and OPERM5) could not be determined from reading the code. Therefore, we only used the 14 tests whose p-values were understood (see Appendix A for a description of these tests).

Below are the results of the DIEHARD tests on the MWC, Xorshift, and Sum-4 generators:

MWC DIEHARD Results

Test Name	Bad p-values	Total p-values	Pct. Bad p-values	No. Failures
Overlapping Sums	10	20	50.0	10
Overall				5
Birthday spacings	4	20	20.0	4
Count-the-1's (specific)	29	500	5.8	4
OQSO	31	560	5.5	3
GCD	3	40	7.5	3
Craps Test 2	3	40	7.5	3
OPSO	26	460	5.7	3
Binary rank (32x32)	2	20	10.0	2
Gorilla	2	20	10.0	2
Parking lot	2	20	10.0	2
Bitstream test	15	400	3.8	1
Binary rank (6x8)	1	20	5.0	1
DNA	33	620	5.3	1
Craps Test 1	0	40	0.0	0
Count-the-1's (stream)	0	20	0.0	0
3D Spheres	0	20	0.0	0
Minimum Distance	0	20	0.0	0
Binary rank (31x31)	0	20	0.0	0
Squeeze	0	20	0.0	0
TOTAL:	**161**	**2880**	**5.6**	**44**

Xorshift DIEHARD Results

Test Name	Bad p-values	Total p-values	Pct. Bad p-values	No. Failures
Overlapping Sums	7	20	35.0	7
Count-the-1's (specific)	34	500	6.8	6
Birthday spacings	4	20	20.0	4
3D Spheres	4	20	20.0	4
Minimum distance	4	20	20.0	4
OQSO	24	560	4.3	3
Binary rank (32x32)	3	20	15.0	3
Count-the-1's (stream)	3	20	15.0	3
OPSO	27	460	5.9	3
Squeeze	3	20	15.0	3
Overall				3
Craps Test 1	2	40	5.0	2
Bitstream test	19	400	4.8	1
Parking lot	1	20	5.0	1
Gorilla	1	20	5.0	1
Craps Test 2	1	40	2.5	1
Binary rank (6x8)	1	20	5.0	1
Binary rank (31x31)	1	20	5.0	1
GCD	1	40	2.5	1
DNA	37	620	6.0	0
TOTAL:	**177**	**2880**	**6.1**	**52**

Sum-4 DIEHARD Results

Test Name	Bad p-values	Total p-values	Pct. Bad p-values	No. Failures
Overlapping Sums	7	20	35.0	7
OPSO	29	460	6.3	4
OQSO	32	560	5.7	4
Count-the-1's (specific)	29	500	5.8	4
Craps Test 2	3	40	7.5	3
Overall				1
Birthday spacings	1	20	5.0	1
DNA	26	620	4.2	1
GCD	1	40	2.5	1
Binary rank (31x31)	1	20	5.0	1
Binary rank (6x8)	1	20	5.0	1
Bitstream	18	400	4.5	0
Squeeze	0	20	0.0	0
Craps Test 1	0	40	0.0	0
3D Spheres	0	20	0.0	0
Minimum Distance	0	20	0.0	0
Count-the-1s (stream)	0	20	0.0	0
Binary rank (32x32)	0	20	0.0	0
Parking lot	0	20	0.0	0
Gorilla	0	20	0.0	0
TOTAL:	**148**	**2880**	**5.1**	**28**

Figure 9.6. Results of DIEHARD test runs on twenty 257-megabyte random number files for each of the MWC, Xorshift, and Sum-4 generators. The number of failing ("Bad") p-values for each test across the twenty files is reported, along with the total number of p-values and the percentage of "bad" ones. The number of overall test failures is presented, based upon the reported scoring criteria. Note: the p-values for the "Overall" test are not tabulated because DIEHARD records all p-values internally and performs its own final test on

them.

Observe that the Sum-4's total number of test failures (at 28) is less than the total number of test failures for both MWC (at 44) and Xorshift (at 52). Also note that the Sum-4's percentage of failing p-values (at 5.1) is slightly less than both MWC's percentage (at 5.6) and Xorshift's percentage (at 6.2).

Lastly, the twenty 257-megabyte files for the MWC, Xorshift, and Sum-4 generators were analyzed by the ENT test program [WAL98]. For each file of each generator, the test reported an entropy of 7.999999 bits per byte.

Thus, with respect to the DIEHARD testing suite against the MWC and Xorshift generators, the Sum-4 cipher exhibits superior randomness properties. All three generators have high entropy.

9.4 Running Time Versus Number of Cells

A good measurement of performance for a cipher would be how fast it can encrypt plaintext and decrypt ciphertext. Because encryption and decryption are the same in the Sum-4 cipher (by XORing with the key-stream), only encryption measurements are presented:

No. of Cells	Running Time (seconds)
32	5.0
64	10.3
96	14.7
128	19.6
160	23.6
192	28.8
224	32.8
256	32.7
288	37.0
320	43.6
352	45.9
384	52.2
512	69.7

Figure 9.7. Running time for encrypting 1 megabyte of plaintext using various cell sizes. These results are measured using a computer program written in the C language with a static plaintext block compiled in.

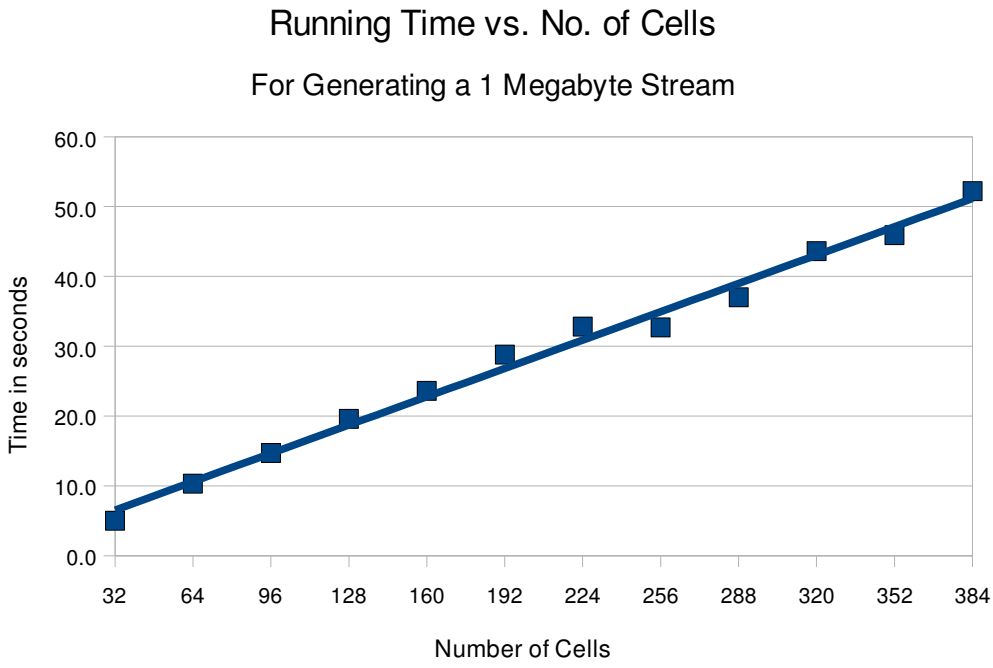

Figure 9.8. Increasing time for encrypting 1 megabyte of plaintext as the number of cells increase. The linear regression line (y = 0.1268x + 2.4818) is shown, having a correlation co-efficient of 0.9951.

These results show that generating a 1-megabyte key-stream from the cipher is very slow. This can be attributed to its software implementation and not necessarily its design. This issue and how it can be resolved is discussed further in chapter 10: Future Work.

9.5 Linear Feedback Shift Register (LFSR) Approximation Testing

A linear feedback shift register (LFSR) is a well-known random number generator that is vulnerable to a known-plaintext attack when used as a key-stream generator. It can be modeled by computing a linear boolean expression over a register of finite size, outputting the bit result, then shifting the output back into that register. For example, consider a 4-bit register (denoted by x_3, x_2, x_1, x_0) initialized to 1011. If we use the equation $x_4 = x_1$ XOR x_0, then the first output bit will be 0. This value is then shifted into x_3, which causes x_3 to be shifted into x_2, etc.; x_0 is discarded entirely. After generating one output bit, the register state will be 0101. A second iteration yields an output of 1 with a register state of 1010. A third iteration yields an output of 1 with a register state of 1101. This action is shown in below in Figure 9.9:

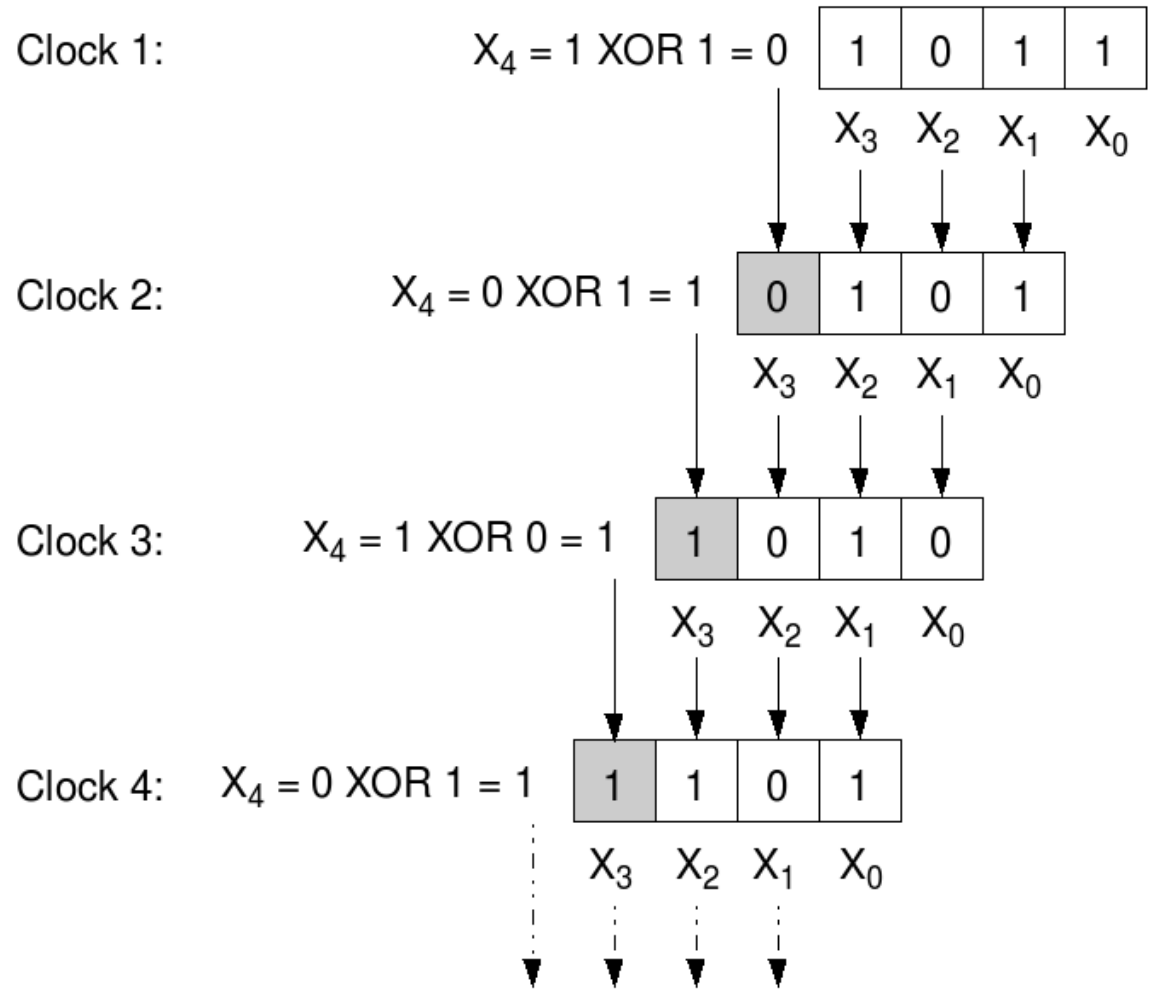

Figure 9.9. The evolution of a linear finite shift register (LFSR). The squares in gray are the registers whose values are outputted.

Massey's algorithm [MAS69] inputs a bit sequence, and outputs an LFSR equation that, when executed, outputs the bit sequence input. Using a known-plaintext attack to recover the key-stream from an LFSR, this algorithm can be used to predict all future key-stream bits, resulting in a major cryptographic vulnerability. Thus, LFSRs are not suitable for use in cryptography. Furthermore, Massey's algorithm can be useful in checking if an LFSR can approximate the key-stream of a stream cipher; if so, then the stream cipher is clearly weak. We shall attempt this against the Sum-4 cipher.

To see if a 256-register LFSR can approximate the Sum-4 cipher, the following steps are performed:

1. Two 256-bit blocks (S_1, S_2) are read in from a file containing a key-stream.
2. S_1 is fed into Massey's algorithm, resulting in an LFSR that produces S_1.
3. The LFSR is used to generate another 256 bits, S_3.
4. The number of matching bits in S_2 and S_3 are counted and outputted.
5. Continue to Step 1, unless key-stream file is exhausted, or process is terminated.

In this testing, we used version 2.10 of the SAGE open-source software (available on the Internet at <http://www.sagemath.org/> as of April 16, 2008) to execute Massey's

algorithm.

The above algorithm counts how many bits out of 256 that an LFSR approximation can accurately predict. Below are the test results for the Sum-4 cipher with 256 cells in each of the control and main CAs:

Total number of bits tested:	158,334,976
Number of matching bits:	79,164,434
Percentage of matching bits:	49.998%
Chi-square:	0.2356
Two-tailed p-value with 1 degree of freedom:	0.6274

Figure 9.10. Results for 256-bit LFSR approximation of the Sum-4 cipher with 256 cells.

Because the LFSRs predicted the output of the Sum-4 cipher approximately 50% of the time, they are no better than simple guessing. According to these results, the 256-cell Sum-4 cipher cannot be approximated by 256-bit LFSRs.

9.6 SAT Solver Testing

Because the cell state in a new generation of a CA is determined by applying a

boolean equation to cells in the previous generation, the entire CA can be expressed as a large system of boolean expressions. As such, it is possible to use a SAT solver application to encode a segment of a discovered key-stream (recovered using a known-plaintext attack) into a boolean expression that represents the CA. This would allow an attacker to compute all cell values in the CA, including the key (which is the initial state). For a strong CA-based cipher, a designer would hope that an attack with a SAT solver would result in an infeasible amount of computation time for a reasonably-sized key.

The specific SAT solver used in this phase is zChaff [MOS01] v2007.3.12. zChaff only operates on files containing boolean expressions in conjunctive normal form (CNF). CNF expressions purely consist of conjunctions of clauses, where the clauses are disjunctions of literals. For example, the expression $AB + \sim C$ (where AB denotes A AND B, $A + B$ denotes A OR B, and ~A denotes NOT A) is not in CNF because it is not valid to distribute the literal C across a conjunction of literals A and B. However, the expression can be converted into CNF using DeMorgan's Law: $(A + \sim C)(B + \sim C)$. In fact, any expression can be converted into CNF.

Rule 30 has the expression $D = A$ XOR $(B + C)$, which is not in CNF. Aside from applying DeMorgan's Law, to convert to CNF we may examine its truth table:

A	***B***	***C***	***D***
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Figure 9.11. The truth table for rule 30, where the output is D *and the inputs are* A*,* B*, and* C*.*

By taking the rows for which $D = 0$, and negating the input literals that equal 1 then ORing them together, we form a single clause of disjuncted literals. Combining all these clauses with ANDs will then result in rule 30's CNF equivalent. Specifically, the first row in figure 9.11 has $D = 0$, so we create a clause with $(A + B + C)$. The last three rows have $D = 0$, so inverting the input literals that are 1 give $(\sim A + B + \sim C)$, $(\sim A + \sim B + C)$, and $(\sim A + \sim B + \sim C)$, for the fifth, sixth, and seventh rows, respectively. Finally, the clauses are all ANDed together to yield the CNF expression: $(A + B + C)(\sim A + B + \sim C)$ $(\sim A + \sim B + C)(\sim A + \sim B + \sim C)$.

While we have demonstrated how to encode a single rule 30 equation into CNF, an

extra step is needed to encode a CA based on rule 30 into a CNF expression. Specifically, the output variable itself must be included so that future generations have something to reference. This is done by constructing another truth table that treats the output variable *D* as yet another input, and outputs a 1 when rule 30 applied on *A*, *B*, and *C* equals *D*:

A	B	C	D	X
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

*Figure 9.12. The truth table for the expression where (*A *XOR (*B + C*)* = D'*) AND (*D = D'*).*

Converting the truth table in Figure 9.12 to CNF using the method described above, we get:

$$(A + B + C + \sim D)(A + B + \sim C + D)(A + \sim B + C + D)(A + \sim B + \sim C + D)(\sim A + B + C + D)$$

$$(\sim A + B + \sim C + \sim D)(\sim A + \sim B + C + \sim D)(\sim A + \sim B + \sim C + \sim D)$$

Figure 9.13. The boolean expression in CNF for the truth table in Figure 9.12.

With the expression in Figure 9.13, we can encode a one-dimensional, one-radius CA based on rule 30 with the following algorithm:

1. Assign *n* unique boolean variables to the initial *n* cells. Place the variables into an order-preserving array named *previousGeneration*.
2. Create an empty order-preserving array named *currentGeneration*.
3. Create a string variable named *cnfString* and initialize it to the empty string.
4. Compute the CNF expression for the current generation of the CA.
    1. Retrieve the three input variables from *previousGeneration* that determine the value of the current cell in the current generation (for the first cell, the left variable is taken as the rightmost variable in *previousGeneration* for CAs with cyclical boundaries; for the last cell, the right variable is taken as the leftmost variable in *previousGeneration* for the same reason) and name them *A*', *B*', and

C '.

2. Assign a new string variable named *tmpR30* a copy the expression in Figure 9.13.
3. Substitute *A*, *B*, and *C* in *tmpR30* for *A*', *B*', and *C* '.
4. Choose a new unique boolean variable, *D*', add it to *currentGeneration* in the current cell position, and substitute it for the variable *D* in *tmpR30*.
5. Combine the *tmpR30* to *cnfString* using an AND operation. Reset *tmpR30* to the empty string.
6. Loop to step 4.1 while there exist more cells to handle in the current generation. Otherwise, set *previousGeneration* to *currentGeneration*, clear *currentGeneration*, and continue to step 5 below.

5. Loop to step 4 while there are more generations to compute. Otherwise, output *cnfString* and terminate.

To illustrate with a small example, consider a 6-cell, one-dimensional, one-radius CA with cyclical boundaries using rule 30 as its transition function. Initializing the algorithm above, we first create the *previousGeneration* array and assign it the values {*A*, *B*, *C*, *D*, *E*, *F*}, which represent the values of the initial state (i.e.: the key). We also

create the *currentGeneration* array then set it to { } (the empty set), and create the *cnfString* string set to "" (the empty string). We begin to compute the next generation, starting with the first cell (cell number 0). We retrieve variables *F*, *A*, and *B* from *previousGeneration*, as they determine the value of the first cell (they are retrieved by using indices (0 – 1 = 5 (mod 6)), 0, and (0 + 1 = 1 (mod 6)), respectively). Next, we copy the equation from Figure 9.13 into the string *tmpR30*, and substitute *A*, *B*, and *C* within it for *F*, *A*, and *B*, respectively. We choose *G* as a new unique boolean variable and substitute it for *D* in *tmpR30* and add *G* to the *currentGeneration* array. We combine *tmpR30* to *cnfString* using an AND operation (though since *cnfString* was empty, it simply becomes a copy of *tmpR30*). A diagram of the current state is in Figure 9.14:

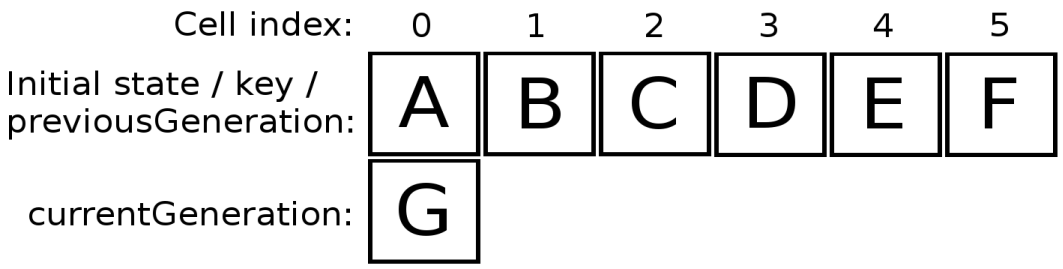

Figure 9.14. The state of the algorithm after it has assigned a boolean variable to the first cell, G*, of the first generation after the initial state (*A*,* B*,* C*,* D*,* E*,* F*). At this step, the* cnfString *variable is set to:*

(F + A + B + ~G)(F + A + ~B + G)(F + ~A + B + G)(F + ~A + ~B + G)(~F + A + B + G)

(~F + A + ~B + ~G)(~F + ~A + B + ~G)(~F + ~A + ~B + ~G)

Continuing on to compute the next cell (cell number 1), we choose the three input variables that determine it by looking up indices (1 – 1 = 0 (mod 6)), 1, and (1 + 1 = 2 (mod 6)) in *previousGeneration*, which correspond to *A*, *B*, and *C*, respectively. These are substituted, like before, along with the next new and unique boolean variable, *H*, resulting in the state shown in Figure 9.15:

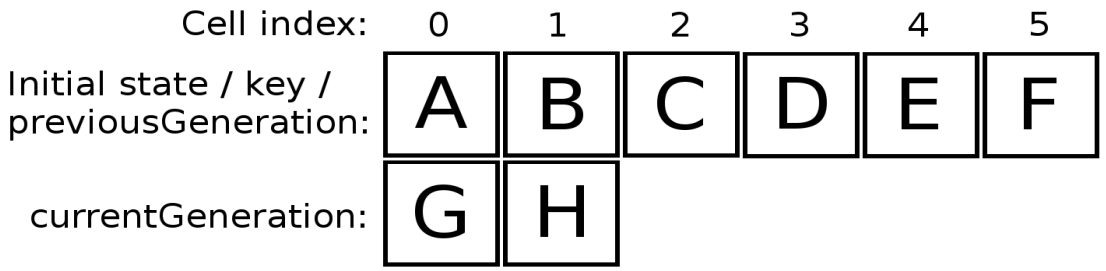

Figure 9.15. The state of the algorithm after it has assign a boolean variable to the second cell, H*, of the first generation after the initial state (*A, B, C, D, E, F*). The CNF expression calculated for this cell is:*

(A + B + C + ~H)(A + B + ~C + H)(A + ~B + C + H)(A + ~B + ~C + H)(~A + B + C + H)

(~A + B + ~C + ~H)(~A + ~B + C + ~H)(~A + ~B + ~C + ~H)

This expression is ANDed to the cnfString *variable to produce:*

(F + A + B + ~G)(F + A + ~B + G)(F + ~A + B + G)(F + ~A + ~B + G)(~F + A + B + G)

(~F + A + ~B + ~G)(~F + ~A + B + ~G)(~F + ~A + ~B + ~G)(A + B + C + ~H)(A + B

+ ~C + H)(A + ~B + C + H)(A + ~B + ~C + H)(~A + B + C + H)(~A + B + ~C + ~H)

(~A + ~B + C + ~H)(~A + ~B + ~C + ~H)

As it can be seen, encoding an *n*-cell CA over *g* generations yields an expression with (*n* * *g*) boolean variables and (*n* * *g* * 8) clauses of disjuncted literals.

While the above method does in fact encode a CA into a CNF expression, processing the expression through a SAT solver would not reveal any useful information, as it would merely find a satisfying assignment of variables for no particular output. An attacker is not interested in this, but rather would want to find the key used to create a key-stream from the CA (or, at the very least, discover some intermediate boolean variables so that the rest of the state can be re-constructed more easily). To do this, the attacker would recover a part of the key-stream using a known-plaintext (or chosen-plaintext) attack. These key-stream bits would then be encoded into the CNF expression so that the SAT solver would find the key that produced those bits. With the key, the attacker could decrypt any ciphertext encrypted with that key, resulting in critical failure of security.

Note that the expression in Figure 9.13 is true if and only if D is the output of A XOR ($B + C$). To modify that expression so that the output is 1, only one term needs to be added:

$$(A + B + C + \sim D)(A + B + \sim C + D)(A + \sim B + C + D)(A + \sim B + \sim C + D)(\sim A + B + C + D)$$

$$(\sim A + B + \sim C + \sim D)(\sim A + \sim B + C + \sim D)(\sim A + \sim B + \sim C + \sim D)\mathbf{(D)}$$

Figure 9.16. The CNF expression for rule 30 for which the output is 1. The end clause in bold is the only addition to the expression in Figure 9.13 necessary to encode the 1 value.

Likewise, the following is the expression in Figure 9.13 modified so that the output is 0:

$$(A + B + C + \sim D)(A + B + \sim C + D)(A + \sim B + C + D)(A + \sim B + \sim C + D)(\sim A + B + C + D)$$

$$(\sim A + B + \sim C + \sim D)(\sim A + \sim B + C + \sim D)(\sim A + \sim B + \sim C + \sim D)\mathbf{(\sim D)}$$

Figure 9.17. The CNF expression for rule 30 for which the output is 0. The end clause in bold is the only addition to the expression in Figure 9.13 necessary to encode the 0 value.

Encoding a CA with the methods described above have been verified to solve for the initial state (key) when processed by a SAT solver.

The Sum-4 cipher uses rule 30 to power its control CA, which is used to provide random bits to drive the main CA's rule selector. A known-plaintext (or chosen-plaintext) attack does not yield any information about the control CA since the key-stream is taken from the main CA. As a result, an attacker could only encode bits into a CNF expression

from the main CA only. Therefore, an extension to the methods above is needed to properly encode the Sum-4 cipher.

To encode the Sum-4 cipher, the control CA is encoded as described above, but with six output variables sampled per generation for encoding the main CA's rule selector:

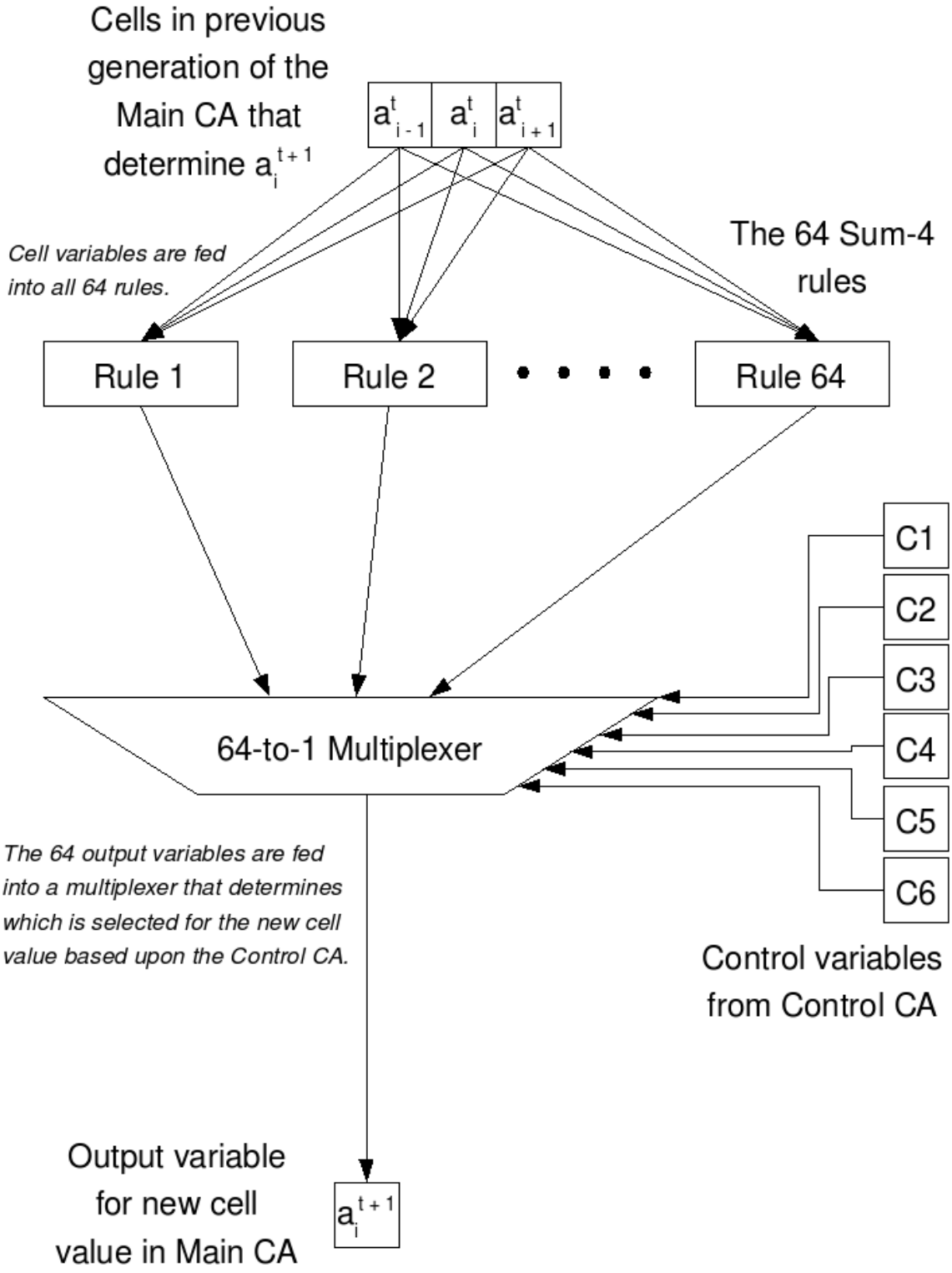
Cells in previous generation of the Main CA that determine a_i^{t+1}
a_{i-1}^t a_i^t a_{i+1}^t
Cell variables are fed into all 64 rules.
The 64 Sum-4 rules
Rule 1
Rule 2
Rule 64
C1
C2
C3
C4
C5
C6
64-to-1 Multiplexer
The 64 output variables are fed into a multiplexer that determines which is selected for the new cell value based upon the Control CA.
Control variables from Control CA
Output variable for new cell value in Main CA
a_i^{t+1}

Figure 9.18. The sum-4 rules being multiplexed into the new value of a Main CA cell based on control variables from the Control CA.

As can be seen in Figure 9.18, the variables from the three cells in the main CA that determine the value of the new cell are substituted into the CNF expressions for all 64 sum-4 rules, yielding 64 output variables. Those output variables are fed into a 64-to-1 multiplexer that selects one output based on six control variables taken from the control CA, mimicking the action of the rule selector.

The CNF expressions for the 64 sum-4 rules were generated using a computer program to generate the truth table for each, then computing the CNF expression from it (see the *boolgenrules.php* script accompanying this thesis paper). The CNF expression for the 64-to-1 multiplexer could not be generated either by hand, nor by a computer program, as the truth table contains 70 variables (64 inputs and 6 control inputs) which would result in 2^{70} rows. Instead, we built a 64-to-1 multiplexer by chaining smaller 2-to-1 multiplexers whose CNF expressions are easily derived. The following illustrates how this is done:

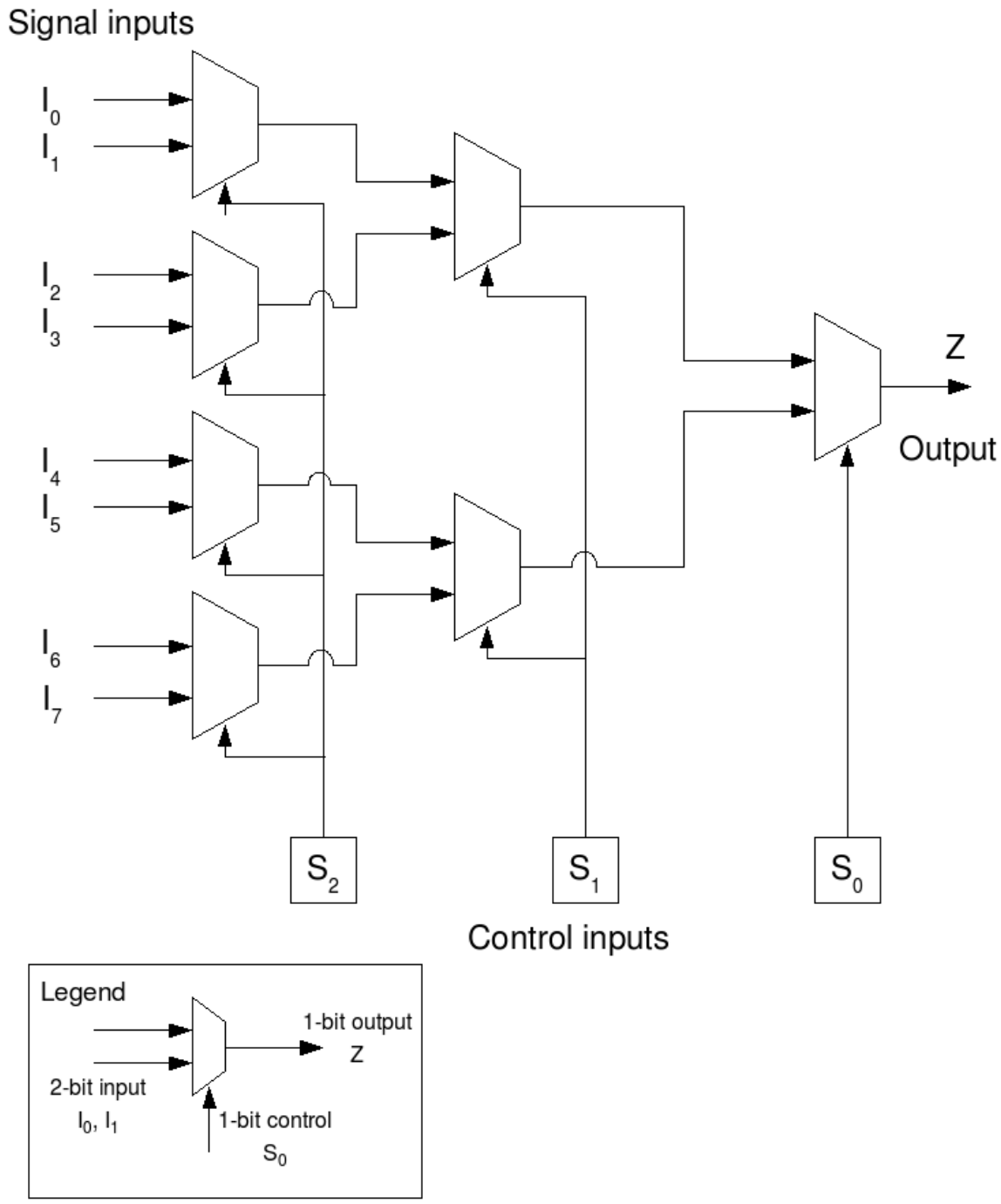

Figure 9.19. An 8-to-1 multiplexer built by chaining 2-to-1 multiplexers together.

Figure 9.19 shows how 2-to-1 multiplexers can be chained together to create a larger 8-to-1 multiplexer. A 64-to-1 multiplexer can easily be created by extending this scheme.

As we have described how the control CA is encoded along with the 64 sum-4 rules, multiplexer, and the method by which the output bits can be encoded (into the main CA), we have fully specified the CNF encoding process of the entire Sum-4 cipher.

An implementation of the Sum-4 CNF encoder has been written in Java that outputs a file format that is used by the zChaff SAT solver [MOS01]. It was used to generate expressions for CA sizes 6 through 20, inclusive, for both the control and main CAs. The size of the CNF file on disk was recorded for each CA size, as well as the amount of memory zChaff used and the time zChaff required to solve it.

CA Size	Size of CNF File (MB)	zChaff Time to Solve	zChaff Time to Solve (seconds)	zChaff Memory Used (MB)	Log_2(Time)	Log_2(Memory)
6	1.1	1s	1	29.6		4.8875
7	1.6	1s	1	48.4		5.5969
8	2.2	3s	3	87.6		6.4529
9	2.8	2s	2	76.0	1.0000	6.2479
10	3.5	6s	6	148.4	2.5850	7.2133
11	4.3	10s	10	212.4	3.3219	7.7306
12	5.2	49s	49	593.1	5.6147	9.2121
13	6.1	1m29s	89	910.0	6.4757	9.8297
14	7.2	2m16s	136	818.1	7.0875	9.6761
15	8.3	5m42s	342	2,462.1	8.4179	11.2657
16	9.4	9m31s	571	3,235.0	9.1573	11.6595
17	10.7	17m51s	1071	2,796.3	10.0647	11.4493
18	12.0	1h11m36s	4296	2,172.5	12.0688	11.0851
19	13.4	3h01m59s	10919	*Unknown*	13.4146	*Unknown*
20	14.9	10h45m08s	38708	*Unknown*	15.2403	*Unknown*

Figure 9.20. Resources needed to analyze the Sum-4 cipher expressed as boolean expressions with various cell sizes using the zChaff SAT solver [MOS01].

The size of the CNF file on disk grows linearly. The memory usage was measured using the Valgrind v3.2.3 software [SEW00], invoked as: *valgrind --tool=memcheck zchaff file.cnf* (Valgrind is a debugging suite whose "memcheck" tool provides information such as dynamic memory usage).

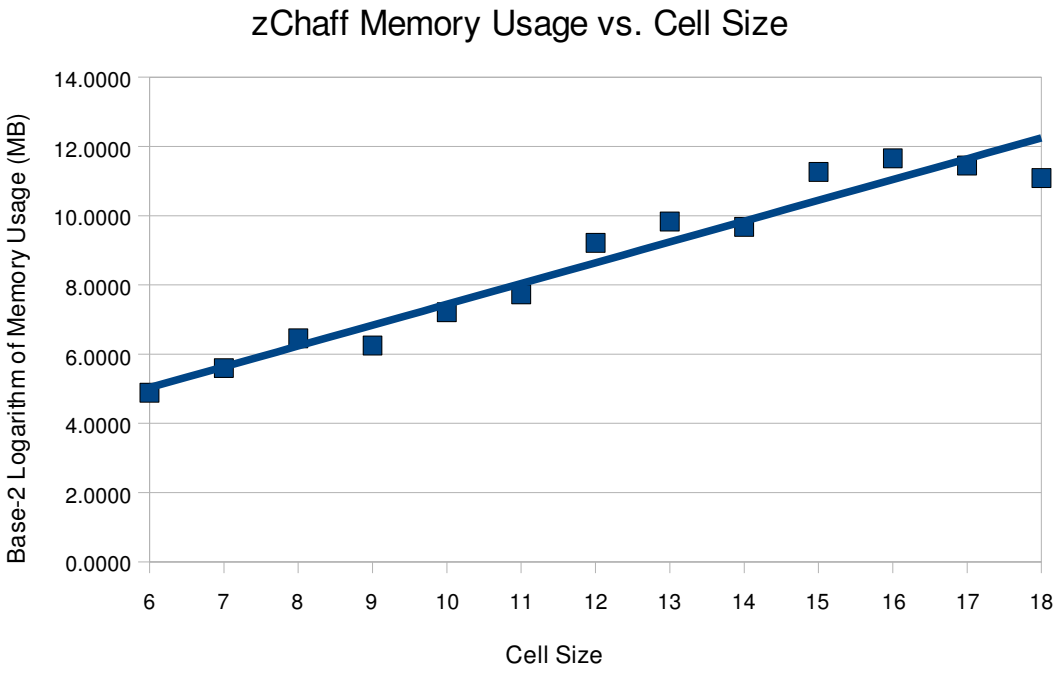

Figure 9.21. Increasing memory usage of SAT solver with respect to cell size of Sum-4 cipher (notice that the Y-axis is the base-2 logarithm of the memory usage in megabytes, thus memory is increasing exponentially, not linearly). The linear regression line (y = 0.6008x + 1.4289) is shown, having a correlation co-efficient of 0.9729.

Memory usage for cell sizes 19 and 20 could not be measured since the Valgrind tool adds extra time overhead. To complete the memory measurement with cell size 18 required over 24 hours. Furthermore, this time was increasing exponentially, suggesting that cell size 19 would require about 3-4 days.

A graph showing the running time required to solve the CNF expressions versus

the number of cells is below:

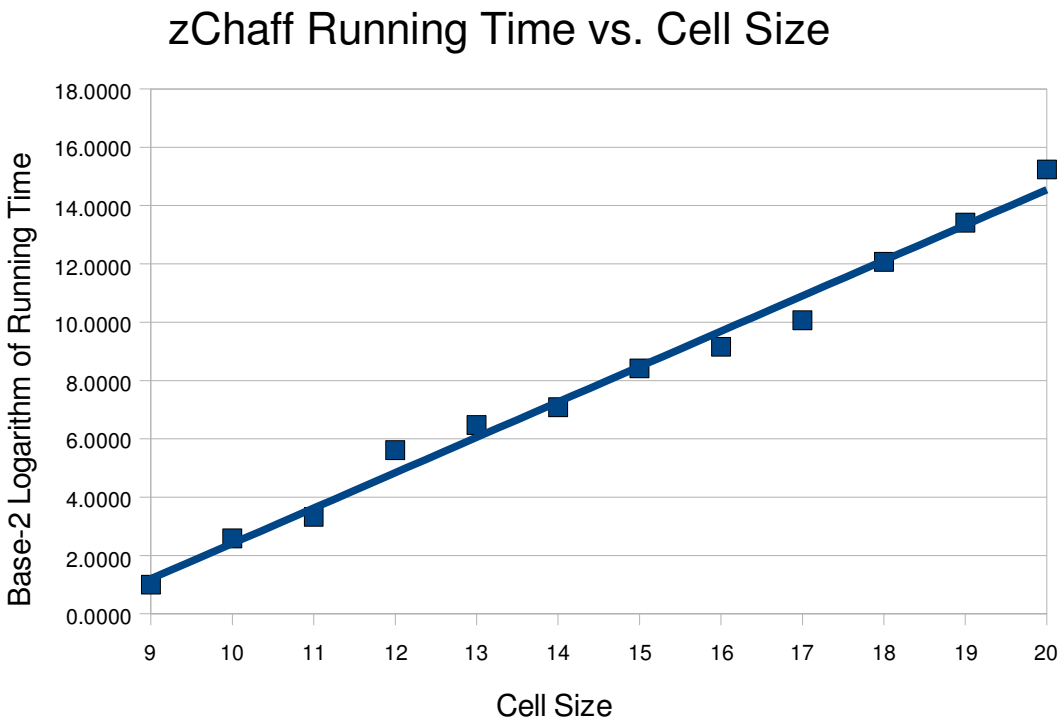

Figure 9.22. Increasing running time of SAT solver with respect to cell size of Sum-4 cipher (notice that the Y-axis is the base-2 logarithm of the running time, thus time is increasing exponentially, not linearly). The linear regression line (y = 1.1717x + (-9.1888)) is shown, having a correlation co-efficient of 0.9898.

As it can be seen, increasing the number of cells in the control and main CAs results in an exponential increase in the time required to find the key. Note that the Y-axis is the

base-2 logarithm of the time in seconds, thus a linear increase in the graph is actually an exponential increase in time. This behavior is desired in a secure system since a small increase in the key size results in a very large increase in effort required by an attacker to find that key. This data is used in section 9.7 below to estimate how large the key should be in order to be secure.

9.7 Effective 128-bit Security

It is suggested in [SCH03] that a key size of 128 bits provides enough brute-force security to preserve secrecy up to 50 years into the future. By comparing the amount of time required to recover the key using a SAT solver with the amount of time required to brute-force all keys, one can compute how many cells are necessary that would force an attacker to use the same amount of resources as a 128-bit brute-force attack.

Below are the results from running a brute-force attack against the Sum-4 cipher:

Cell Size	Time (Java program)	Time (C program)	Log2(Time [C program]) (in secs)
6	0s	0s	-
7	1s	0s	-
8	1s	0s	-
9	4s	1s	0.0000
10	19s	3s	1.5850
11	1m23s	14s	3.8074
12	6m21s	1m04s	6.0000
13	29m34s	4m58s	8.2192
14	2h21m55s	22m57s	10.4273
15	11h21m12s	1h49m25s	12.6806
16	*Unknown*	8h07m43s	14.8368

Figure 9.23. Computation time needed to brute-force a Sum-4 cipher with various cell sizes using computer programs written in Java and C.

Notice that the brute-force implementation using the C language is much faster than the Java language implementation. For this reason, we will only use the results from the C program.

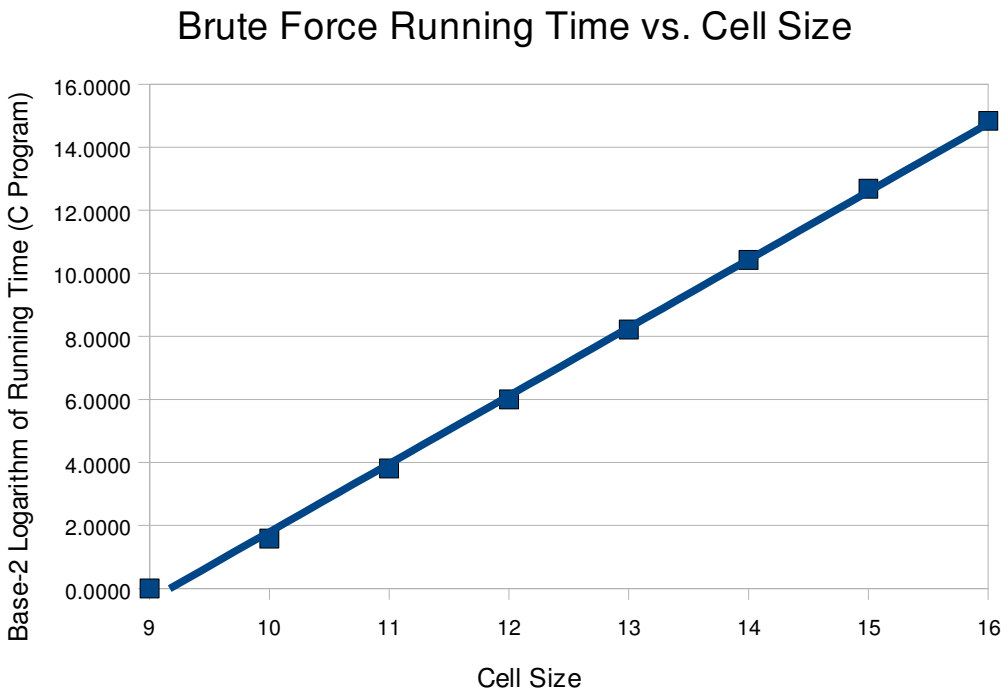

Figure 9.24. Increasing running time of brute-force attack against the Sum-4 cipher using a computer program written in the C language. The linear regression line (y = 2.1597x + (-19.801668)) is shown, having a correlation co-efficient of 0.9994.

To show the effectiveness of the SAT attack versus the brute-force attack, we combine the graphs from Figures 9.21 and 9.23:

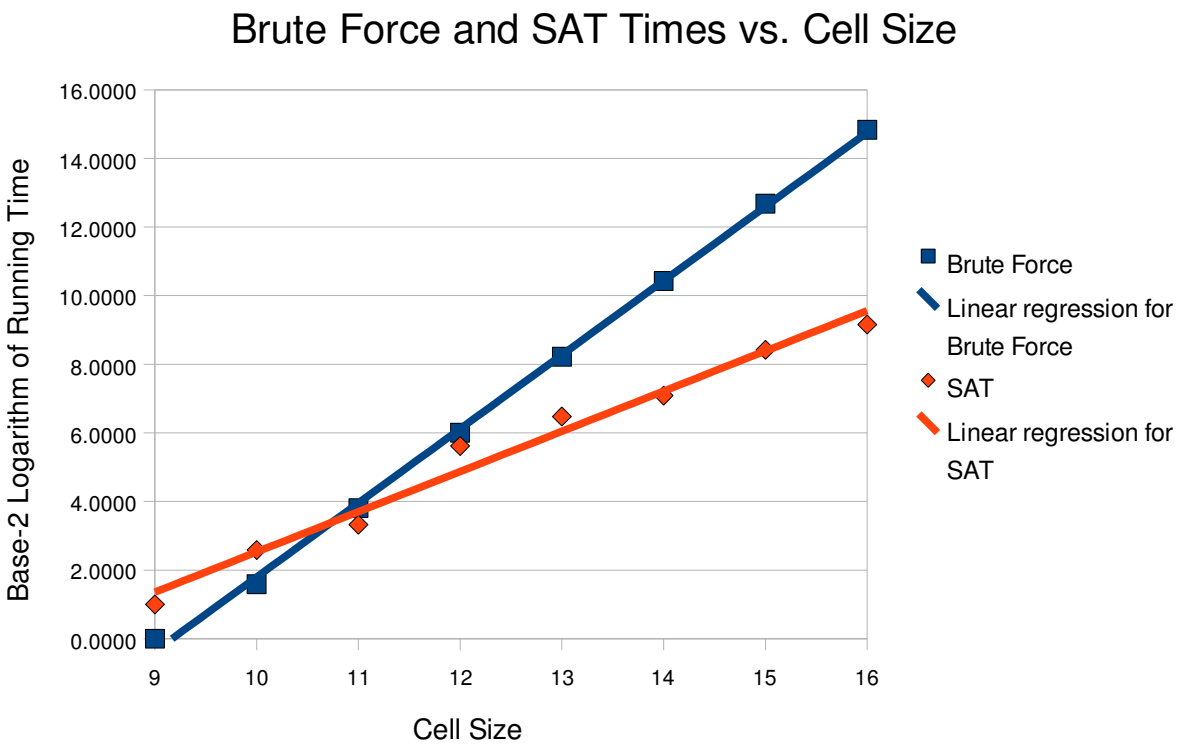

Figure 9.25. The combination of the graphs from Figure 9.22 and 9.24 showing that the SAT attack is more efficient than brute-force.

By calculating a linear regression for the SAT and brute-force results, we can estimate information for larger cell sizes. We can then extrapolate how many cells are necessary to ensure that the more-efficient SAT attack will take as much time as a brute force attack over 128 bits.

The linear regression line for the SAT attack, using time information for cell sizes 9 through 16, inclusive, is:

$$y = 1.1717x + (-9.1888)$$

The linear regression line for the brute-force attack (using the C program) for the same cell sizes is:

$$y = 2.1597x + (-19.8017)$$

Therefore:

$$y = 2.1597(128) + (-19.8017)$$

$$y = 256.6399$$

$$256.6399 = 1.1717x + (-9.1888)$$

$$x = 226.8744$$

$$x = 227$$

Thus, it is estimated that 227 cells for each of the control and main CAs will require the same running time for the SAT attack as a brute-force attack on 128 bits.

9.8 Comparison of Running Times of 128-bit AES with Effective 128-bit Sum-4 Cipher

To see the difference of speed between the Sum-4 cipher with 128-bits of effective security and the 128-bit AES cipher [NIS01c], the running times for encrypting 10, 50, and 100 megabyte plaintexts were recorded. The AES implementation from libgcrypt [KOC98] version 1.2.4 was called from a program implemented in the C language:

Plaintext Size (MB)	AES (secs)	Sum-4 (secs)
10	0.1s	5m42.9s
50	0.6s	27m52.0s
100	1.2s	55m11.8s

Figure 9.26. The speed of encrypting plaintext blocks for 128-bit AES and Sum-4 with a 227-cell configuration.

As it can be seen, the Sum-4 cipher's software implementation is extremely slow. This is due to the fact that several optimizations could not be done due to time constraints. Currently, the implementation stores one cell value per memory location, which is much less efficient than storing multiple cells per location. This also increases load on the computer's memory controller, which results in a significant bottleneck. The current implementation also is not parallel, which is unfortunate since cellular automata

are highly parallel (all cells can be updated simultaneously since they only depend upon the previous generation). Furthermore, cellular automata can be translated naturally into hardware, though this is beyond the scope of the thesis and outside of the expertise of the author. These improvements are noted in chapter 10: Future Work.

9.9 Number of Unique Streams

A strong stream cipher will generate unique streams with respect to unique keys. If multiple keys result in the same key-stream, then an attacker has a higher chance at finding one key that generates that stream, as the effective key space would be reduced.

For small cell sizes, the unique streams resulting from the Sum-4 cipher have been tabulated. For each cell size below, the key space is brute-forced. For each key, a key-stream consisting of (*2 x cell size*)-bits is generated and tested for uniqueness against key-streams from other keys (all within the same cell size). The number of unique key-streams are divided by the theoretical number of key-streams ($2^{2 \times cell\ size}$) to give a percentage of unique streams for that cell size.

Cell Size	No. Possible Streams	No. Unique Streams	Pct. Unique
6	4,096	862	21.0%
7	16,384	4,348	26.5%
8	65,536	13,515	20.6%
9	262,144	49,264	18.8%
10	1,048,576	170,790	16.3%
11	4,194,304	593,684	14.2%
12	16,777,216	1,765,457	10.5%
13	67,108,864	5,981,877	8.9%

Figure 9.27. Low percentage of unique streams from the Sum-4 cipher.

As it can be seen in the above figure, the Sum-4 cipher does not produce many unique streams. In order for the cipher to be practical, this must be fixed. This is noted in chapter 10: Future Work.

10. FUTURE WORK

As with any good research, there are ideas to expand upon and techniques to improve.

Firstly, as pointed out in section 9.9, the Sum-4 cipher does not produce a large percentage of unique key-streams. This significant vulnerability in the security of the cipher was not discovered until late in the research process. Due to time constraints, it could not be rectified. However, we believe that it is a solvable problem. It may be possible to combine the cipher with fast, non-cryptographic pseudo-random number generators such as the MWC or Xorshift generators mentioned in section 9.3. MWC and Xorshift can be implemented very efficiently in both software and hardware, and have very high execution speed. In theory, these generators could be incorporated into the cipher design without much extra overhead.

Secondly, we believed since the initial cipher design that it is plausible that multiple cells per generation could be sampled in order to generate multiple key-streams from the same CAs. Work needs to be done to show that this is possible without sacrificing the security of the cipher; an attacker could perform a known-plaintext attack to recover parallel streams, but should not be able to discover enough information to predict future bits from any stream. If this design idea is successful, then n parallel

streams could be generated from the same amount of computation, or the throughput of the cipher could be increased *n* times (where *n* is the number of cells that can be safely sampled in parallel). This idea, too, might be implemented by combining fast, non-cryptographic generators into the cipher design. Unfortunately, this task was also not pursued due to time constraints.

Thirdly, cycle lengths could be studied empirically for small cell sizes. Short cycle lengths with respect to cell sizes would indicate a problem with the cipher. If a plaintext longer than the cycle length were encrypted, the key-stream would repeat itself, degenerating into the vulnerable Vigenere cipher.

Fourth, the cipher's speed might be improved by replacing the control CA with a fast, non-cryptographic pseudo-random number generator. One would have to study the randomness of its output and research what vulnerabilities, if any, might arise from such a combination.

Fifth, a better software implementation of the Sum-4 cipher could be made. The current implementation does not incorporate any parallelism whatsoever, even though cellular automata are naturally highly parallel. Furthermore, the current implementation stores one cell value per memory location, which causes it to be memory-bound. The execution speed is largely regulated by the speed of the computer's memory controller. This can be greatly improved by storing multiple cell values per memory location and

operating upon all of them simultaneously in the processor's registers. These improvements could speed up execution by many factors—enough to reach (or possibly surpass) the speed of AES.

Sixth, cellular automata can be very efficiently implemented in hardware since all cells can be updated synchronously each generation with only one clock cycle. Unfortunately, hardware design and implementation is outside the expertise of the author.

Seventh, it would seem wise that a practical implementation of the cipher should "prime the pump" as mentioned in [KAM04]; this involves transitioning an n-bit CA n times after startup and discarding the output so that the initial key is properly mixed into the CA state. This concept should be studied more closely to see if more than n initial bits should be discarded.

Lastly, the Sum-4 cipher could be analyzed using the TestU01 random number testing suite [LEC07], as it is mentioned in [LEC07] that the DIEHARD suite has several shortcomings, some of which overlook statistical weaknesses in the Xorshift generator [PAN05].

11. CONCLUSION

In this paper, we have provided an in-depth literature search in chapters 2 through 8 covering all published papers arising from Wolfram's break-through idea to use cellular automata in stream ciphers [WOL86]. In chapter 9, we presented a novel design for a stream cipher based on cellular automata called Sum-4 and studied its performance and security properties. In chapter 10, we presented ideas for future work that can make the Sum-4 cipher both more secure and more efficient.

It is our hope that the work done in this thesis can either one day be used in a practical system, or inspire future cryptographers to create practical systems using versatile cellular automata.

APPENDIX A: DIEHARD Test Descriptions

Below are the descriptions for all the tests used in DIEHARD, taken directly from the testing suite itself [MAR98]:

Birthday Spacings: Choose m birthdays in a "year" of n days. List the spacings between the birthdays. Let j be the number of values that occur more than once in that list, then j is asymptotically Poisson distributed with mean m^3/(4n). Experience shows n must be quite large, say n>=2^18, for comparing the results to the Poisson distribution with that mean. This test uses |n=2^24 and m=2^10, so that the underlying distribution for j is taken to be Poisson with lambda=2^30/(2^26)=16. A sample of 200 j's is taken, and a chi-square goodness of fit test provides a p value. The first test uses bits 1-24 (counting from the left) from 32-bit integers in the specified file. The file is closed and reopened, then bits 2-25 of the same integers are used to provide birthdays, and so on to bits 9-32. Each set of bits provides a p-value, and the nine p-values provide a sample for a KSTEST.

GCD: Let the (32-bit) RNG produce two successive integers u,v. Use Euclids algorithm to find the gcd, say x, of u and v. Let k be the number of steps needed to get x. Then k is approximately binomial with p=.376 and n=50, while the distribution of x is very close to Pr(x=i)=c/i^2, with c=6/pi^2. The gcd test uses ten million such pairs u,v to see if the resulting frequencies of k's and x's are consistent with the above distributions. Congruential RNG's---even those with prime modulus---fail this test for the distribution of k, the number of steps, and often for the distribution of gcd values x as well.

Gorilla: This is the GORILLA test, a strong version of the monkey tests that I developed in the 70's. It concerns strings formed from specified bits in 32-bit integers from the RNG. We specify the bit position to be studied, from 0 to 31, say bit 3. Then we generate 67,108,889 (2^26+25) numbers from the generator and form a string of 2^26+25 bits by taking bit 3 from each of those numbers. In that string of 2^26+25 bits we count the number of 26-bit segments that do not appear. That count should be approximately normal with mean 24687971 and std. deviation 4170. This leads to a normal z-score and hence to a p-value. The test is applied for

each bit position 0 (leftmost) to 31. (Some older tests use Fortran's 1-32 for most- to least- significant bits. Gorilla and newer tests use C's 0 to 31.) This is the BINARY RANK TEST for 31x31 matrices. The leftmost 31 bits of 31 random integers from the test sequence are used to form a 31x31 binary matrix over the field {0,1}. The rank is determined. That rank can be from 0 to 31, but ranks< 28 are rare, and their counts are pooled with those for rank 28. Ranks are found for 40,000 such random matrices and a chisquare test is performed on counts for ranks 31,30,28 and <=28. (The 31x31 choice is based on the unjustified popularity of the proposed "industry standard" generator x(n) = 16807*x(n-1) mod 2^31-1, not a very good one.)

Binary rank (32x32): This is the BINARY RANK TEST for 32x32 matrices. A random 32x32 binary matrix is formed, each row a 32-bit random integer. The rank is determined. That rank can be from 0 to 32. Ranks less than 29 are rare, and their counts are pooled with those for rank 29. Ranks are found for 40,000 such random matrices and a chisquare test is performed on counts for ranks 32,31, 30 and <=29.

Binary rank (6x8): This is the BINARY RANK TEST for 6x8 matrices. From each of six random 32-bit integers from the generator under test, a specified byte is chosen, and the resulting six bytes form a 6x8 binary matrix whose rank is determined. That rank can be from 0 to 6, but ranks 0,1,2,3 are rare; their counts are pooled with those for rank 4. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks <=4, 5 and 6.

Bitstream: The file under test is viewed as a stream of bits. Call them | b1,b2,... . Consider an alphabet with two "letters", 0 and 1 and think of the stream of bits as a succession of 20-letter "words", overlapping. Thus the first word is b1b2...b20, the second is b2b3...b21, and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2^21 overlapping 20-letter words. There are 2^20 possible 20 letter words. For a truly random string of 2^21+19 bits, the number of missing words j should be (very close to) normally distributed with mean 141,909 and sigma 428. Thus (j-141909)/428 should be a standard normal variate (z score) that leads to a uniform [0,1) p value. The test is repeated twenty times.

OPSO (Overlapping-Pairs-Sparse-Occupancy): The OPSO test considers 2-letter words from an alphabet of 1024 letters. Each letter is determined by a specified ten bits from a 32-bit integer in the sequence to be tested. OPSO generates 2^21 (overlapping) 2-letter words (from 2^21+1 "keystrokes") and counts the number of missing words---that is, 2-letter words which do not appear in the entire sequence. That count should be very close to normally distributed with mean 141,909, sigma 290. Thus (missingwrds-141909)/290 should be a standard normal variable. The OPSO test takes 32 bits at a time from the test file and uses a designated set of ten consecutive bits. It then restarts the file for the next designated 10 bits, and so on.

OQSO (Overlapping-Quadruples-Sparse-Occupancy): The test OQSO is similar, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the test file, elements of which are assumed 32-bit random integers. The mean number of missing words in a sequence of 2^21 four- letter words, (2^21+3 "keystrokes"), is again 141909, with

sigma = 295. The mean is based on theory; sigma comes from extensive simulation.

DNA: The DNA test considers an alphabet of 4 letters: C,G,A,T, determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, so that as in OPSO and OQSO, there are 2^20 possible words, and the mean number of missing words from a string of 2^21 (over- lapping) 10-letter words (2^21+9 "keystrokes") is 141909. The standard deviation sigma=339 was determined as for OQSO by simulation. (Sigma for OPSO, 290, is the true value (to three places), not determined by simulation.

Count-the-1's (stream): This is the COUNT-THE-1's TEST on a stream of bytes. Consider the file under test as a stream of bytes (four per 32 bit integer). Each byte can contain from 0 to 8 1's with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte: 0,1,or 2 yield A 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E.

Thus we have a monkey at a typewriter hitting five keys with vari- ous probabilities (37,56,70,56,37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (over- lapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chisquare test: Q5-Q4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.

Count-the-1's (specific): This is the COUNT-THE-1's TEST for specific bytes. Consider the file under test as a stream of 32-bit integers. From each integer, a specific byte is chosen , say the left- most: bits 1 to 8. Each byte can contain from 0 to 8 1's, with probabilitie 1,8,28,56,70,56,28,8,1 over 256. Now let the specified bytes from successive integers provide a string of (overlapping) 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's, in that byte: 0,1,or 2 ---> A, 3 ---> B, 4 ---> C, 5 ---> D, and 6,7 or 8 ---> E. Thus we have a monkey at a typewriter hitting five keys with with various probabilities: 37,56,70, 56,37 over 256. There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-

letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chisquare test: Q5-Q4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.

Parking lot: In a square of side 100, randomly "park" a car---a circle of radius 1. Then try to park a 2nd, a 3rd, and so on, each time parking "by ear". That is, if an attempt to park a car causes a crash with one already parked, try again at a new random location. (To avoid path problems, consider parking helicopters rather than cars.) Each attempt leads to either a crash or a success, the latter followed by an increment to the list of cars already parked. If we plot n: the number of attempts, versus k: the number successfully parked, we get a curve that should be similar to those provided by a perfect random number generator. Theory for the behavior of such a random curve seems beyond reach, and as graphics displays are not available for this battery of tests, a simple characteriz ation of the random experiment is used: k, the number of cars successfully parked after n=12,000 attempts. Simulation shows that k

should average 3523 with sigma 21.9 and be approximate to normally distributed. Thus (k-3523)/21.9 should serve as a standard normal variable, which, converted to a p uniform in [0,1), provides input to a KSTEST based on a sample of 10.

Minimum Distance: It does this ten times: choose n=8000 random points in a square of side 10000. Find d, the minimum distance between the (n^2-n)/2 pairs of points. If the points are truly inde- pendent uniform, then d^2, the square of the minimum distance should be (very close to) exponentially distributed with mean .995 . Thus 1-exp(-d^2/.995) should provide a p-value and a KSTEST on the resulting 10 values serves as a test of uni- formity for those samples of 8000 random points in a square.

3D Spheres: Choose 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest point. Then the volume of the smallest such sphere is (very close to) exponentially distributed with mean 120pi/3. Thus the radius cubed is exponential with mean 30. (The mean is obtained by extensive

simulation). The 3DSPHERES test gener- ates 4000 such spheres 20 times. Each min radius cubed leads to a uniform variable by means of 1-exp(-r^3/30.), then a KSTEST is done on the 20 p-values.

Squeeze: Random integers are floated to get uniforms on [0,1). Starting with k=2^31=2147483647, the test finds j, the number of iterations necessary to reduce k to 1, using the reduction k=ceiling(k*U), with U provided by floating integers from the file being tested. Such j's are found 100,000 times, then counts for the number of times j was <=6,7,...,47,>=48 are used to provide a chi-square test for cell frequencies.

Overlapping Sums (OSUM): Integers are floated to get a sequence U(1),U(2),... of uni- form [0,1) variables. Then overlapping sums, S(1)=U(1)+...+U(100), S2=U(2)+...+U(101),... are formed. The S's are virtually normal with a certain covariance mat- rix. A linear transformation of the S's converts them to a sequence of independent standard normals, which are converted to uniform variables for a KSTEST.

Craps 1: It plays 200,000 games of craps, counts the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean 200000p and variance 200000p(1-p), and p=244/495. Throws necessary to complete the game can vary from 1 to infinity, but counts for all>21 are lumped with 21. A chi-square test is made on the no.-of-throws cell counts. Each 32-bit integer from the test file provides the value for the throw of a die, by floating to [0,1), multiplying by 6 and taking 1 plus the integer part of the result.

Craps 2: This is the CRAPS TEST with different dice. Each die value is determined by the rightmost three bits of the 32-bit random integer; values 1 to 6 are accepted, others rejected. As in the first test, 200,000 games of craps are played, counting the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean 200000p and variance 200000p(1-p), and p=244/495. Throws necessary to complete the game can vary from 1 to infinity, but counts for all>21 are lumped with 21. A chi-square test is made on the no.-of-throws cell counts.

Overall: In response to requests, we have provided a list of all the p-values produced by the tests you have chosen for this run. The individual p-values are supposed to be uniform in [0,1), but they are not necessarily independent. So even though we have applied a KSTEST to the accumulated p-values, the result is not necessarily---even if your file contains truly random bits---uniform in [0,1). But it is probably pretty close, so take that last p-value with a grain of salt. In particular, there may be some values so close to 0 or 1 that the tests they came from should be applied several more times, or new, related tests should be undertaken.

APPENDIX B: Developer Manual

The following describes how to compile all software associated with this Master's thesis paper. Note that all software was developed and tested on Linux. While it probably can work with little or no changes on other platforms, no guarantees can be made.

1.) The four PHP scripts (*autoresults.php*, *autotester.php*, *boolgenrules.php*, and *lfsrtest.php*) do not need to be compiled.

2.) To compile *CASAT.java*, ensure that Java v5 is installed and type at a command prompt:

```
$ javac CASAT.java
```

3.) The four C programs (*aes.c*, *mwc.c*, *s4ca.c*, and *xorshift.c*) can be compiled with a single Makefile. aes.c requires that libgcrypt v.1.2.0 or later (and its development files) be installed. It is available as a pre-compiled package on many Linux distributions, including Ubuntu (or it can be compiled from source from <http://www.gnupg.org/> as of April 16, 2008.)

Ensure that a working gcc environment is available and that the four C programs are in the same directory as the Makefile and type:

```
$ make
```

APPENDIX C: User Manual

The following describes how to use all software associated with this Master's thesis paper. Note that all software was developed and tested on Linux. While it probably can work with little or no changes on other platforms, no guarantees can be made.

autotester.php

Description: This script runs the DIEHARD suite on all the *.rnd files in the specified directory. If 'test_directory' is the name of the directory containing *.rnd files, then the results are put in 'test_directory_results'.

Usage: `$ php autotester.php test_directory`

autoresults.php

Description: Processes the results from the autotester.php script. When run on the

result directory created by that script, this script will compile and tabulate the results into a file called 'results.txt' in the result directory.

Usage: `$ php autoresults.php results_directory`

boolgenrules.php

Description: This script generates the CNF expressions for one-dimensional, one-radius CA rules in the $rules array. It outputs the expressions into a Java code string so that it may be directly imported. This script was used to generate the 'ruleCNFs' array in CASAT.java.

Usage: `$ php boolgenrules.php`

lfsrtest.php

Description: This script takes 256 bits from the Sum-4 key stream (given by the user as

the program argument) and calculates a 256-bit linear finite shift register (LFSR) that is capable of generating those bits. It then uses the LFSR to generate another 512 kilobits and compares them to the next 512 kilobits in the key stream file. It tabulates how many bits match and how many differ. If an LFSR can approximate the key stream of a generator, then this is strong evidence that it is weak.

Tests performed on the Sum-4 cipher suggest that a 256-bit LFSR can predict its output no better than random guessing, as close to 50% of the bits match. This is a good sign of security.

This script requires SAGE to be installed. SAGE is an open-source mathematics program that contains an implementation of J.L. Massey's algorithm to construct an LFSR based on output bits (in section 3 of "Shift-register synthesis and BCH decoding," IEEE Trans. Inform. Theory,

vol. IT-15, pp. 122-127, Jan. 19 69.). SAGE is available at:

<http://modular.math.washington.edu/sage/>.

This script will periodically save its progress to 'file.rnd.lfsr' so that if interrupted, it can automatically resume where it left off. That file will contain the batch number (which is of no concern to the user), number of similar bits, and total number of bits tested in a format such as "12|6294636|12582912".

To use this script, edit the $SAGE_DIR variable to point to the directory

where SAGE is installed, then execute:

Usage: `$ php lfsrtest.php file.rnd`

CASAT.java

Description: This program outputs a boolean expression for the Sum-4 stream cipher. The expression will be in conjunctive normal form (CNF) and is suitable for processing by a SAT solver such as zChaff. It is also able to calculate the number of unique key streams from a certain cell size. Lastly, it can brute-force a cipher so that timing information can be obtained with respect to CA size.

Usage: To generate a CNF file with 18 cells (for control and main CAs *each*) and with 36 output bits:

```
$ java CASAT --cells=18 --outputFactor=2 \
    --output=cell18_output36.cnf
```

To calculate the number of unique key streams arising from 8 cells and 16 output bits:

```
$ java CASAT --cells=8 --outputFactor=2 --test-
uniqueness
```

To brute-force a 14-cell cipher with 14 output bits (NOTE: this option is obsoleted by the s4ca.c program!):

```
$ time java CASAT --cells=14 --outputFactor=1 --
brute-force
```

aes.c

Description: This program encrypts a set plaintext block with 128-bit AES. It is useful for benchmarking purposes only.

Usage (to encrypt 10MB of plaintext and put it into file.enc):

```
$ ./aes 10 file.enc
```

mwc.c

Description: This is an implementation of the Multiple with Carry (MWC) random number generator, published by G. Marsaglia (2003) in "Xorshift RNGs", Journal of Statistical Software, vol. 8, no. 14, pp. 1-6.

Usage: `$ ./mwc 10 tenmegs.rnd`

s4ca.c

Description: This is the (relatively) optimized version of the cipher. It can:

- run the cipher with any number of cells.
- create files of any size containing the key stream.
- provide time measurements for generating key streams.
- brute-force a cipher with any number of cells.

Usage:

To output 5 megabytes into the file 'stream.rnd' using 256 cells for the control and main CAs each:

```
$ ./s4ca --cells=256 --output-megs=5 \
     --output-file=stream.rnd
```

To brute-force a cipher with 14 cells and 28 output bits:

```
$ ./s4ca --cells=14 --output-bits=28 --brute-force
```

xorshift.c

Description: This is an implementation of the XOR-shift random number generator, published by G. Marsaglia (2003) in "Xorshift RNGs", Journal of Statistical Software, vol. 8, no. 14, pp. 1-6.

Usage: `$ ./xorshift 20 twentymegs.rnd`

REFERENCES

[ALV06] Alvarez, G., and Li, S., “Some Basic Cryptographic Requirements for Chaos-Based Cryptosystems,” *International Journal of Bifurcation and Chaos,* vol. 16, no. 8, p. 2129-2151, 2006.

[ANO94] Anonymous, "Thank you Bob Anderson," <http://cypherpunks.venona.com/date/1994/09/msg00304.html>, 1994, Retrieved on April 21, 2008.

[BAD02] Badr, A., “An Alternative Cellular Automata Cryptogram,” *Studies in Informatics and Control*, vol. 11, p. 339-347, 2002.

[BAO03] Bao, F., "Cryptanalysis of a New Cellular Automata Cryptosystem," *Information Secu*rity *and Privacy, 8th Australasian Conference, ACISP 2003,* Wollongong, Australia, July 9-11 2003, vol. 2727 of *Lecture Notes in Computer Science,* Springer.

[BAO04] Bao, F., “Cryptanalysis of a Partially Known Cellular Automata Cryptosystem,” *IEEE Transactions on Computers*, vol. 53, no. 11, p. 1493-1497, 2004.

[BLA97] Blackburn, S., Murphy, S., and Paterson, K., “Comments on 'Theory and Applications of Cellular Automata in Cryptography',” *IEEE Transactions on Computers*, vol. 46 no. 5, p. 637-638, 1997.

[BOU03] Bouvry P., Seredynski F., and Zomaya A., "Application of Cellular Automata for Cryptography," *Lecture Notes in Computer Science*, vol. 3019, p. 447-454, 2003.

[BOU05] Bouvry P., Klein G., and Seredynski F., "Weak Key Analysis and Micro-Controller Implementation of CA Stream Ciphers," *Lecture Notes in Artificial Intelligence*, vol. 3684, p. 910-915, 2005.

[CAT96] Cattell, K. and Muzio, J., “Analysis of One-Dimensional Linear Hybrid Cellular Automata over GF(q)”, *IEEE Transactions on Computers*, vol. 45, pp. 782-792, 1996.

[DAE93] Daemen, J., Govaerts, R., and Vandewalle, J., "A Framework for the Design of

One-Way Hash Functions Including Cryptanalysis of Damgard's One-Way Function Based on a Cellular Automaton," *Advances in Cryptology - Asiacrypt 91*, *Lecture Notes in Computer Science*, vol. 739, p. 82-96, 1993.

[DAM90] Damgård, I., "A Design Principle for Hash Functions," *Advances in Cryptology - Crypto 89*, *Lecture Notes in Computer Science*, vol. 435, p. 416-427, 1990.

[DAN04] D'Antonio, M. and Delzanno, G., "SAT-Based Analysis of Cellular Automata," *Lecture Notes in Computer Science*, vol. 3305, p. 745-754, 2004.

[DAS99] Dasgupta, P., Chattopadhyay, S., and Sengupta, I., "An ASIC for Cellular Automata Based Message Authentication," *12th International Conference on VLSI Design*, 1999.

[DEL06] Delgado, D., Vidal, D., and Hernandez, G., "Evolutionary Design of Pseudorandom Sequence Generators Based on Cellular Automata and its Applicability in Current Cryptosystems," *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, July 08-12, 2006, Seattle, Washington, USA.

[DIE06] Dierks, T. and Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.1," *The Internet Engineering Task Force RFC 4346*, 2006.

[ENC05] Encinas, L., del Rey, A., and Sanchez, G., “A CA-based protocol to authenticate images,” *The 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, July 13th, 2005, Paris, France.

[FRA04] Franti, E., Slav, C., Balan, T., and Dascalu, M., "Design of Cellular Automata Hardware for Cryptographic Applications," *Semiconductor Conference, 2004. CAS 2004 Proceedings. 2004 International*, Oct. 4-6 2004, vol. 2, p. 463-466.

[FSF91] The Free Software Foundation, "GNU General Public License, Version 2," <http://www.gnu.org/licenses/gpl.html>, 1991, Retrieved April 18, 2008.

[GUA87] Guan, P., “Cellular Automaton Public-Key Cryptosystem,” *Complex Systems*, vol. 1, p. 51-57, 1987.

[GUT93] Gutowitz, H., “Cryptography with Dynamical Systems,” *Proceedings of the NATO Advanced Study Institute: Cellular Automata and Cooperative Systems*, p. 237-274, 1993.

[HOR89a] Hortensius, P., McLeod, R., and Card, H., "Parallel random number generation for VLSI systems using cellular automata." IEEE Transactions on Computers, vol. 38, issue 10, p. 1466-1473, 1989.

[HOR89b] Hortensius, P., McLeod, R., Pries, W., Miller, M., and Card, C., "Cellular automata-based pseudorandom number generators for built-in self-test," IEEE Transactions on Computer-Aided Design, vol. 8, issue 8, p. 842-859, 1989.

[IEE04] IEEE, "IEEE Standard for Information Technology--Telecommunications and Information Exchange Between Systems--Local and Metropolitan Area Networks--Specific Requirements--Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 6: Medium Access Control (MAC) Security Enhancements," *IEEE Std 802.11i-2004*, 2004.

[JOS06] Joshi, P., Mukhopadhyay, D., and RoyChowdhury, D., “Design and Analysis of a

Robust and Efficient Block Cipher Using Cellular Automata," *The 20th International Conference on Advanced Information Networking and Applications (AINA 2006), IEEE Computer Society*, vol. 2, p. 67–71, 2006.

[KAM04] Kaminsky, A., "Cellular Automata Based Stream Ciphers Lecture Notes," <http://www.cs.rit.edu/~ark/lectures/casc01/casc01.pdf>, 2004, Retrieved on April 21, 2008.

[KOC97] Koc, C.K. and Apohan, A.M., "Inversion of Cellular Automata Iterations", *IEEE Proceedings in Computers and Digital Technology*, vol. 144, p. 279-284, 1997.

[KOC98] Koch, W., et al., "Libgcrypt," <http://www.gnupg.org/>, 1998, Retrieved April 16, 2008.

[LAF96] Lafe, O., "Data Compression and Encryption Using Cellular Automata Transforms," *IEEE International Joint Symposia on Intelligence and Systems (IJSIS '96)*, 234–241, 1996.

[LEC07] L'Ecuyer, P. and Simard, R., "TestU01: a C library for empirical testing of

random number generators," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, August 2007.

[LEE06] Lee, M., Hong, D, and Kim, D., "Cryptanalysis of Mukherjee-Ganguly-Chaudhuri's Message Authentication Scheme," *Computational Intelligence and Security, 2006 International Conference on*, vol. 2, p. 1311-1314, Nov. 3-6 2006.

[LEN03] Len, R., Encinas, A., Encinas, L., White, S.H., del Rey, A.M., Sanchez, G.R., and Ruiz, I.V., "Wolfram Cellular Automata and Their Cryptographic Use as Pseudorandom Bit Generators," *International Journal of Pure Applied Mathematics*, vol. 4, p. 87-103, 2003.

[LIU05] Liu, J., Cheng, X., and Wang, X., "Cryptanalysis of a Cellular Automata Cryptosystem", *Lecture Notes in Artificial Intelligence*, vol. 3802, p. 49-54, 2005.

[MAR98] Marsaglia, G., "DIEHARD: Battery of Tests of Randomness" <http://stat.fsu.edu/~geo/diehard.html>, 1998, Retrieved April 16, 2008.

[MAR03a] Marsaglia, G., "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no.

14, p. 1-6., 2003.

[MAR03b] Maranon, G., Encinas, G., Encinas, L., del Rey, A., and Sanchez, G., "Graphic Cryptography with Pseudorandom Bit Generators and Cellular Automata," *Lecture Notes in Artificial Intelligence*, vol. 2773, p. 1207-1214, 2003.

[MAR05] Maranon, G., Encinas, L., and del Rey, A., "A New Secret Sharing Scheme for Images Based on Additive 2-Dimensional Cellular Automata," *Lecture Notes in Computer Science*, vol. 3522, p 411-418, 2005.

[MAR06] Marconi, S. and Chopard B., "Discrete Physics, Cellular Automata and Cryptography," *Lecture Notes in Computer Science*, vol. 4173, p. 617-626, 2006.

[MAS69] Massey, J., "Shift-register synthesis and BCH decoding," *IEEE Trans. Inform. Theory*, vol. IT-15, p. 122-127, 1969.

[MAT98] Matsumoto, M., "Simple Cellular Automata as Pseudorandom m-Sequence Generators for Built-in Self-test," *ACM Transactions on Modeling and Computer*

Simulation (TOMACS), vol. 8, no. 1, p. 31-42, 1998.

[MEI91] Meier, W. and Staffelbach, O., "Analysis of Pseudorandom Sequences Generated by Cellular Automata," *Advances in Cryptology: Eurocrypt '91*, *Lecture Notes in Computer Science*, vol. 547, pp. 186-199, Springer-Verlag, Heidelberg, 1991.

[MIH97a] Mihaljevic. M., "An Improved Key Stream Generator Based on the Programmable Cellular Automata," *Proc. of ICICS'97*, *Lecture Notes in Computer Science*, vol. 1334, p. 181-191. Springer-Verlag, 1997.

[MIH97b] Mihaljevic, M., "Security Examination of a Cellular Automata Based Pseudorandom Bit Generator using an Algebraic Replica Approach," *Applied Algebra, Algorithms and Error Correcting Codes – AAECC 12*, *Lecture Notes in Computer Science*, vol. 1255, p. 250-262, 1997.

[MIH98a] Mihaljevic, M., Zheng, Y., and Imai, H., "A Cellular Automaton Based Fast One-way Hash Function Suitable for Hardware Implementation," *Lecture Notes in Computer Science*, vol. 1431, p. 217-233. Springer-Verlag, 1998.

[MIH98b] Mihaljevic, M., Zheng, Y., and Imai, H., "A Fast and Secure Stream Cipher Based on Cellular Automata over GF(q)", *IEEE GLOBECOM '98*, Sydney, Australia, Nov. 1998, pp. 3250-3255.

[MIH98c] Mihaljevic, M., Zheng, Y., and Imai, H., "A Fast Cryptographic Hash Function Based on Linear Cellular Automata over GF(q)," *Global IT Security*, IFIP, pp. 96-107, 1998 (*Proc. 14th IFIP Int. Conf. on Information Security at 15th World Comput. Congress - IFIP/SEC '98*, Vienna - Budapest, Sept. 1998).

[MOS01] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. “Chaff: Engineering an Efficient SAT Solver,” *Proceedings of the 38th Design Automation Conference (DAC '01)*, 2001.

[MUK02] Mukherjee, M., Ganguly, N., and Chaudhuri, P.P., “Design of Cellular Automata Based Message Authentication” *International Conference on Computer Communication*, Mumbai, 2002.

[MUK04] Mukhopadhyay, D. and Roy-Chowdhury, D., "Cellular Automata: An ideal Candidate for a Block Cipher," *In the Pmceedings of First International Conference on Distributed Computing and Internet Technology ICDCIT 2004, Lecture Notes in Computer Science,* vol. 3347*,* 2004.

[MUK07] Mukhopadhyay, D. and RoyChowdhury, D., "Theory of a Class of Complemented Group Cellular Automata and its Application to Cryptography," *Journal of Cellular Automata*, vol. 2, no. 3, p. 243-271, 2007.

[NAN94] Nandi, S., Kar, B., and Chaudhuri, P., "Theory and Applications of Cellular Automata in Cryptography," *IEEE Transactions on Computers*, vol. 43, no. 12, p. 1346-1357, 1994.

[NIS99] United States Government, National Institute of Standards and Technology (NIST), “Data Encryption Standard,” *Federal Information Processing Standards (FIPS) Publication 46-3*, October 25, 1999.

[NIS01a] United States Government, National Institute of Standards and Technology

(NIST), “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,” *NIST Special Publication 800-22*, May 15, 2001.

[NIS01b] United States Government, National Institute of Standards and Technology (NIST), “Security Requirements for Cryptographic Modules,” *Federal Information Processing Standards (FIPS) Publication 140-2*, May 25, 2001.

[NIS01c] United States Government, National Institute of Standards and Technology (NIST), “Advanced Encryption Standard (AES),” *Federal Information Processing Standards (FIPS) Publication 197*, November 26, 2001.

[NIS07] United States Government, National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family," *Federal Register*, vol. 72, no. 212, Office of the Federal Register, November 2, 2007.

[OLI04] Oliveira, G., Coelho, A., Monteiro, L., "Cellular Automata Cryptographic Model Based on Bi-directional Toggle Rules," *International Journal of Modern Physics C*, vol.

15, no. 8, p. 1061-1068, 2004.

[PAN05] Panneton, F. and L'Ecuyer, P., “On the xorshift random number generators,” *ACM Transactions on Modeling and Computer Simulation*, vol. 15, no. 4, p. 346-361, October 2005.

[POP05] Popovici, A. and Popovici, D., "A Generalization of the Cellular Automata Rule-30 Cryptoscheme," *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 158-164, 2005.

[REY05] del Rey, A., "Design of a Cryptosystem Based on Reversible Memory Cellular Automata," *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, p. 482-486, June 27-30 2005.

[RUB04] Rubio, C., Encinas, L., White, S., del Rey, A., and Sanchez, G., “The Use of Linear Hybrid Cellular Automata as Pseudorandom Bit Generators in Cryptography,” *Neural Parallel Sci. Comput*., vol. 12, p. 175-192, 2004.

[SCH03] Schneier, B. and Ferguson, N., Practical Cryptography, John Wiley & Sons, p. 37, 2003.

[SEN02] Sen, S., Shaw, C., Chowdhuri, R., Ganguly, N., and Chaudhuri, P., "Cellular Automata Based Cryptosystem (CAC)," *Proceedings of the Fourth International Conference of Information and Communications Security (ICICS02)*, pp. 303-314, Dec. 2002.

[SEN03] Sen, S., Hossain, S., Islam, K., Roy Chowdhuri, D., Chaudhuri, P., "Cryptosystem Design for Embedded System Security," *VLSI Design*, 2003.

[SER03] Seredynski, F., Bouvry, P., and Zomaya, A., "Cellular Programming and Symmetric Key Cryptography Systems," *Lecture Notes in Computer Science*, vol. 2724, p. 1369-1381, 2003.

[SER04] Seredynski, F., Bouvry, P., and Zomaya, A., "Cellular Automata Computations and Secret Key Cryptography," *Parallel Computing*, vol. 30, no. 5-6, p. 753-766, May-Jun 2004.

[SER05] Seredynski, M. and Bouvry, P., “Block Cipher Based on Reversible Cellular Automata,” New Generation Computing, v.23 n.3, p.245-258, September 2005.

[SEW00] Seward, J., et al., "Valgrind," <http://www.valgrind.org/>, 2000, Retrieved April 16, 2008.

[SHA49] Shannon, C., “Communication Theory of Secrecy Systems.” *Bell Systems Technical Journal*, vol. 28, p. 656-715, 1949.

[SHA01] Shaw, C., Chatterji, D., Maji, P., Sen, S., Roy, B., and Chaudhuri, P., “A Pipeline Architecture for Encompression (Encryption + Compression) Technology,” *Proceedings of the International Conference on VLSI Design*, India, p. 454–459, January 2001.

[SHA02] Shackleford, B., Tanaka, M., Carter, R., and Snider, G., “FPGA Implementation of Neighborhood-of-four Cellular Automata Random Number Generators,” *Proceedings of the 2002 ACM/SIGDA 10th International Symposium on Field-programmable Gate Arrays*, Feb 24-26, 2002, Monterey, California, USA.

[SIP96] Sipper, M. and Tomassini, M., “Co-evolving Parallel Random Number Generators,” *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, p. 950-959, Sept 22-26, 1996.

[SRE03] Srebrny, M. and Such, P., “Encryption using Two-dimensional Cellular Automata with Applications,” *Artificial Intelligence and Security in Computing Systems*, Kluwer Academic Publishers, Norwell, MA, 2003.

[SRI95] Srisuchinwong, B., York, T., and Tsalides, P., "A Symmetric Cipher using Autonomous and Non-autonomous Cellular Automata," *Global Telecommunications Conference, 1995. GLOBECOM '95., IEEE*, vol. 2, p. 1172-1177, Nov 14-16, 1995.

[SUN07] Sung, J., Hong, D., and Hong, S., “Cryptanalysis of an Involutional Block Cipher using Cellular Automata,” *Information Processing Letters*, vol. 104., no. 5., p. 183-185, 2007.

[SZA06a] Szaban, M., Seredynski, F., and Bouvry, P., "Collective Behavior of Rules for Cellular Automata-based Stream Ciphers," *Evolutionary Computation, 2006. CEC 2006.*

IEEE Congress on , p. 179-183, Jul 16-21, 2006.

[SZA06b] Szaban, M., Seredynski, F., and Bouvry, P., "Evolving Collective Behavior of Cellular Automata for Cryptography," *Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*, p. 799-802, May 16-19, 2006.

[TOM95] Tomassini, M., "A survey of genetic algorithms," *Annual Reviews of Computational Physics*, vol 3, p. 87-118., World Scientific, 1995.

[TOM00a] Tomassini, M., Sipper, M., Perrenoud, M., "On the Generation of High-Quality Random Numbers by Two-Dimensional Cellular Automata," *IEEE Transactions on Computers*, vol. 49, no. 10, p.1146-1151, 2000.

[TOM00b] Tomassini, M. and Perrenoud, M., "Stream Ciphers with One- and Two-Dimensional Cellular Automata," *Parallel Problem Solving from Nature – PPSN VI*, *Lecture Notes in Computer Science*, vol. 1917, pp. 722-731, Springer, 2000.

[TOM01] Tomassini, M. and Perrenoud, M., "Cryptography with Cellular Automata"

Applied Soft Computing Journal, vol. 1, no. 2, p. 151-160, 2001.

[VER26] Vernam, G., “Cipher printing telegraph systems for secret wire and radio telegraphic communications,” Journal of the IEEE, vol 55, p. 109-115, 1926.

[WAL98] Walker, J., "ENT: A Pseudorandom Number Sequence Test Program", <http://www.fourmilab.ch/random/>, 1998, Retrieved April 21, 2008.

[WOL86] Wolfram, S., “Cryptography with Cellular Automata,” *Advances in Cryptology: Crypto '85 Proceedings, Lecture Notes in Computer Science*, vol. 218, p. 429-432, Springer-Verlag,1986.

[XUE04] Xuelong, Z., Jiwen, W., Manwu, X., and Fengyu, L., “High-quality Pseudo-random Sequence Generator Based on One-dimensional Extended Cellular Automata,” *Proceedings of the 3rd International Conference on Information Security*, Nov 14-16, 2004, Shanghai, China.

[XUE05] Xuelong, Z., Qianmu, L., Manwu, X., and Fengyu, L., "A Symmetric Cryptography based on Extended Cellular Automata," *Systems, Man and Cybernetics,*

2005 IEEE International Conference on, vol. 1, pp. 499-503, Oct. 10-12, 2005.

[ZHA02] Zhang, C. and Hua, L., "Reconfigurable Pipelined Cellular Automata Array for Cryptography," *Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference on*, vol. 2, p. 1213-1217, Jun 29-Jul 1, 2002.